



华章科技

CENGAGE  
Learning

**G** 游戏开发技术系列丛书

**BEGINNING GAME PROGRAMMING** (Third Edition)

# 游戏编程入门

(原书第3版)



(美) Jonathan S. Harbour 著  
陈征 傅鑫 等译



机械工业出版社  
China Machine Press



# 游戏编程入门 (原书第3版)

## BEGINNING GAME PROGRAMMING (Third Edition)

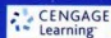
本书向初出茅庐的游戏开发人员展示了将游戏思想从概念转变为现实的方法。本书涵盖了使用 DirectX 编写代码创建 2D 和 3D 游戏所需的所有技能,而读者只需基本掌握 C++ 语言即可学会。游戏的每个元素都按部就班地在书中讲授——从学习如何创建简单的 Windows 程序,到使用关键的 DirectX 组件来渲染 2D 和 3D,再到给游戏添加声音。通过学习本书讲授的技能,读者可以开发出属于自己的用于构建将来的游戏项目的游戏库。本书在每章结束时新增了测验题和项目以便帮助读者实践新学到的技能!本书最后部分通过创建一个完整的、全功能的游戏来实践所介绍的新技能。

### 作者简介:

**Jonathan S. Harbour** 是 University of Advancing Technology ( [www.uat.edu](http://www.uat.edu) ) 的副教授。他编写过许多编程方面的书籍,并且教授这方面的课程,包括 DirectX、Allegro、Python、Lua、DarkBASIC、多游戏者网络、XNA Game Studio 以及 Java。他还教授包括任天堂 GameCube ( Dolphin SDK )、任天堂 GBA 和任天堂 DS ( DevKitPro )、Sony PlayStation 2 ( TOOL )、Xbox 360 ( XNA SDK ) 以及手机 ( Java ME ) 在内的专有课程。他著有《Advanced 2D Game Development》。他目前忙于一个涉及多线程的新项目。

### 配套光盘内容:

- 书中所有示例项目的源代码
- 附录 D 中额外的着色器示例
- 诸如 Mappy 和 Pro Motion 这样的有用工具的 30 天试用版



[www.cengageasia.com](http://www.cengageasia.com)

客服热线: (010) 88378991, 88361066  
购书热线: (010) 68326294, 88379649, 68995259  
投稿热线: (010) 88379604  
读者信箱: [hzsj@hzbook.com](mailto:hzsj@hzbook.com)

华章网站 <http://www.hzbook.com>

CENGAGE  
Learning™



网上购书: [www.china-pub.com](http://www.china-pub.com)

封面设计: 杨宇梅

## 游戏开发技术系列丛书

上架指导: 计算机 / 程序设计

ISBN 978-7-111-32860-5



9 787111 328605

定价: 55.00元 (附光盘)

BEGINNING GAME PROGRAMMING (Third Edition)

# 游戏编程入门

(原书第3版)



Jonathan S. Harbour 著  
陈征 傅鑫 等译



机械工业出版社  
China Machine Press

分享  
PDF

本书从基本的 Windows 编程开始, 为游戏编程入门者介绍了使用 DirectX 在 Windows 下编写游戏所需的基础知识。读者将学习到把思想转化为现实所需的技术, 比如 2D、3D 图形的绘制、背景滚动、处理游戏输入、音效、碰撞检测等。

本书语言简练, 适合有志于进入游戏编程世界且有一定 C++ 编程基础的初学者阅读, 也适合作为社会培训机构的培训教材。

Jonathan S. Harbour; Beginning Game Programming, Third Edition

EISBN: 978-1-435-45427-9

Copyright © 2010 by Course Technology, a part of Cengage Learning.

Original edition published by Cengage Learning. All Rights reserved. 本书原版由圣智学习出版公司出版。版权所有, 盗印必究。

China Machine Press is authorized by Cengage Learning to publish and distribute exclusively this simplified Chinese edition. This edition is authorized for sale in the People's Republic of China only (excluding Hong Kong, Macao SAR and Taiwan). Unauthorized export of this edition is a violation of the Copyright Act. No part of this publication may be reproduced or distributed by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

本书中文简体字翻译版由圣智学习出版公司授权机械工业出版社独家出版发行。此版本仅限在中华人民共和国境内(不包括中国香港、澳门特别行政区及中国台湾)销售。未经授权的本书出口将被视为违反版权法的行为。未经出版者预先书面许可, 不得以任何方式复制或发行本书的任何部分。

Cengage Learning Asia Pte. Ltd.

5 Shenton Way, # 01-01 UIC Building, Singapore 068808

本书封面贴有 Cengage Learning 防伪标签, 无标签者不得销售。

封底无防伪标均为盗版

版权所有, 侵权必究

本书法律顾问 北京市展达律师事务所

本书版权登记号: 图字: 01-2010-5149

图书在版编目(CIP)数据

游戏编程入门(原书第3版)/(美)哈本(Harbour, J.S.)著;陈征等译.—北京:机械工业出版社, 2011.1

(游戏开发技术系列丛书)

书名原文: Beginning Game Programming, Third Edition

ISBN 978-7-111-32860-5

I. 游… II. ①哈… ②陈… III. 游戏—软件设计 IV. TP311.5

中国版本图书馆 CIP 数据核字(2010)第 254220 号

机械工业出版社(北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑: 秦 健

北京市荣盛彩色印刷有限公司印刷

2011 年 1 月第 1 版第 1 次印刷

186mm×240mm·19 印张

标准书号: ISBN 978-7-111-32860-5

ISBN 978-7-89451-808-8(光盘)

定价: 55.00 元(附光盘)

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991; 88361066

购书热线: (010) 68326294; 88376949; 68995259

投稿热线: (010) 88379604

读者信箱: hzjsj@hzbook.com



## 译者序

游戏是计算机世界永恒的主题。玩游戏容易，写游戏就不那么简单了。很多初学者想自己编写游戏软件，却发现大量鸿沟摆在面前，想一个一个越过却找不着门道。

本书从基本的 Windows 编程开始，为游戏编程入门者介绍了使用 DirectX 在 Windows 下编写游戏所需的基础知识。读者将学习到把思想转化为现实所需的技术，例如 2D 和 3D 图形的绘制、背景滚动及处理游戏输入、音效及碰撞检测等。

本书语言简练，适合有志于进入游戏编程世界的初学者阅读。无论读者是否具备 DirectX 的知识，甚至哪怕只是个中学生，只要有一定的 C++ 编程基础，就能从中获益。

参加本书翻译的人员有：陈征、傅鑫、孟庆麟、戴锋、许瑛琪、王开年、易小丽、陈婷、管学岗、王新彦、金惠敏等。

由于时间紧迫，加之译者水平有限，错误在所难免，恳请广大读者批评指正。

译者

2010 年 9 月



## 序

“我想做一个游戏设计师，我该怎么做才能得到这份工作？”这是我在面试或者与学生交谈时经常被问到的问题。在我和我的团队离开时，一位极具天赋的少年的父母曾经和我搭讪。我通常的回答是：“那么，你设计过什么？”这时，绝大多数人会给我一个冗长的解释，说他们有许多伟大的想法，但就是缺少让自己梦想成真的团队。我对此的反应是，跟他解释和我一起工作的人都有伟大的想法，但只有很少一部分是设计师。

我并不是有意要这么苛刻，但是，不会有任何成功的公司愿意给初出茅庐的人一个开发团队，给他超过 18 个月的时间和几百万美元的预算，却无需任何概念上的证明。这就是现实。Sid Meier（传奇游戏设计师，我很荣幸能和他一起在 Firaxis Games 工作）这样的人之所以能够鹤立鸡群是因为他不但能够采纳别人的想法还能将其转化为有趣的东西。当然，Sid 现在有了大团队来做他的项目，但他总是以相同的方式开始做项目，用他能找到或者自己制作的美工作品和声音作品来粗略制作出一组原型。正是这些粗糙的概念表达使得与创作过程不相关的人们可以立即看到一个想法的有趣之处，他从而也获得了预算和团队。每位崭露头角的设计师会记下笔记然后问道：“Sid 会怎么做？”

于是，一本像这样的图书就显得弥足珍贵。我认识 Jonathan 已有两年，当时我在游戏开发者大会的书店里看到了本书最初的版本。我的一个程序员朋友帮我从大量的相似书籍里把它挑了出来。他认为这是一本写得很好的书，而且认为对 DirectX 的强调将非常适用于我们在 Firaxis 所做的工作。另外一个同伴提及他曾经读过 Jonathan 关于 Game Boy Advance 编程的著作，而且印象深刻。我觉得他们给了我极好的建议，我在通读本书时获得了极大的享受。在阅读的时候，我注意到 Jonathan 是我们的游戏——Sid Meier 的《Civilization III》的粉丝。于是我联系了他，因为我从事过 Civ 系列大量的工作，并从此与他保持联系。

像这样的一本书，漂亮就漂亮在它所有的借口都扫光了。它完美地介绍了游戏编程。它将携手你完成编写 C 代码并利用 DirectX 来实现游戏这一似乎复杂的过程。你在全然未知中已经得到了一个完全可用的框架来实现你的想法。你甚至可以创建自己的美工效果和声音的工具，让它们给游戏上妆。换言之，读者将得到所有的制作原型及证明自己并不仅仅是只有伟大想法的人所需的工具。相信我，经过这关键的一步，你就会成为那些想在这一行业中找到工作的人中的佼佼者。你将具备脱颖而出的能力，而目前有非常多的人都想在游戏开发中分一杯羹，这种能力至关重要。

那么，Sid 会怎么做？当他去年在制作 Sid Meier 的《Railroads!》原型的时候，他用 C 写了整个原型。他那时没有美工人员（他们那时都忙于其他项目），于是他学习了一种 3D 美工程序，自己做美工，然后把这些用到游戏中，他经常使用文本标签来保证游戏者知道游戏中涉及的东西。他使用来自前一个 Firaxis 游戏和 Internet 上的音频文件，四处点缀，增强游戏者的体验。他在很短的时间内创造出了一些东西，这些东西让发行商和其他人了解了这个游戏会有多么有趣。



## 欢迎前来冒险

欢迎来到游戏编程的冒险地带！笔者热衷游戏和游戏编程有许多年了，我和你一样对这一曾经神秘的主题充满激情。游戏，这里指的是 PC 游戏，曾经只出现在奇客的国度里——这是一个冒险家探索无限想象世界，然后自己努力创建出相似世界的地方。同时，在真实世界中，人们过着正常的生活。

那么为什么我们宁可当宅人呢？是因为我们觉得看屏幕上的像素更有趣吗？没错！

但有些人眼里的像素却是另外一些人的幻想世界或外太空冒险。最早的游戏只是屏幕上推来推去的一堆像素而已。但即使在过去玩那些原始的游戏时，我们的想像力还是经常会给我们带来更多意识不到的细节。

那么，你的激情是什么？或者说你最喜欢的游戏类型是什么？是经典的射击街机、幻想冒险、实时战略游戏，角色扮演游戏，还是与运动相关的游戏？我希望读者在阅读本书的同时能够在脑海中设计一个你自己的游戏，随着逐步深入到每一章，都应想象要如何创建这个游戏。本书并不想通过一堆带补丁的代码清单和下一步该如何操作的建议，给读者一个对游戏开发的“暖融融的”感觉。我希望读者读完最后一章时能获得一种完成的喜悦感。从某种程度上说这是一本完备的书籍，读者可从中学到制作自己早期的游戏项目的实用知识。在这里，读者学到的东西将足以用来编写一个完整的、质量足够好的游戏，并且能够满怀信心地与他人分享。

本书将教授读者如何用 C++ 语言编写 DirectX 代码。游戏编程是一个极富挑战的主题，不仅难以精通，而且难以入门。本书通过使用 C++ 和 DirectX 这两种行业工具来揭开游戏编程的神秘面纱。读者将学到使用 Windows 和 DirectX 来渲染 2D 和 3D 图形的方法。

我们将学习编写简单的 Windows 程序的方法。以此为基础，我们将学习 DirectX 的关键组成部分：Direct3D、DirectSound、DirectInput 和 D3DXSprite。本书将教会读者如何利用这些关键的 DirectX 组件以及如何编写通俗易懂的简单代码。在这个过程中，我们会将所有从每章收集来的新知识放到一个游戏库中，以便在将来的游戏项目中重用。在学习了编写简单游戏所需的所有知识之后，读者将看到创建一个横向卷轴射击游戏的方法！

### 从何处开始

我的游戏开发哲学是让普通程序员有的放矢。本书将从一开始就进入正题，而不是讲解标准 C++ 库中的每个函数调用。所以，如果读者对 C++ 还不熟悉的话，那么现在就开始学习它吧。可以肯定的是，有大量和本书所用的语言一样强大（甚至有过之）的伟大产品可以为我们所用。例如 Blitz Basic（见 Maneesh Sethi 所著的《Game Programming for Teens》一书）和 DarkBASIC（见 Jonathan Harbour 和 Joshua Smith 所著的《DarkBASIC Pro Game Programming》第 2 版），这是两种能向用户提供完整工具包的游戏开发工具，包括编译器、编辑器、游戏库 / 引擎以及生成无需任何类型的运行库和独立的 Windows/DirectX 游戏的能力。如果读者是个 C++ 语言的新手，或者

而他自己一个人做了这一切，就如他在车库里工作的那些“旧时光”一样。

那么，你会怎么做？如果你想在业内获得一份游戏设计师的工作，甚至只是想制作一个酷酷的游戏来教你的女儿学数学，你就应该购买这本书。投入进来，完成练习，开始开发你自己的游戏库——Sid 还在用一些还是 Commodore 64 时代的代码。让想像力自由飞翔，然后找到将想法转换为人们可以实际享受的东西。

无论做什么，只要去做就可以。这是设计师的学习和成长之路，也是实现游戏设计师梦想的金钥匙。如果 Sid 还不是 Sid，也没有那么多可以运用自如的工具，那么他可能已经开始在设计这些工具了。

Barry E. Caudill

执行制作人

Firaxis Games

2K Games

Take 2 Interactive





# 目 录

译者序  
序  
欢迎前来冒险

## 第一部分 Windows和DirectX 游戏编程引言

第1章 Windows初步 .....	2
1.1 Windows编程概述 .....	2
1.1.1 认识Windows .....	3
1.1.2 Windows消息机制 .....	4
1.1.3 多任务 .....	5
1.1.4 多线程 .....	6
1.1.5 事件处理 .....	7
1.2 DirectX快速概览 .....	8
Direct3D是什么 .....	9
1.3 Windows程序基础 .....	9
1.3.1 创建第一个Win32项目 .....	10
1.3.2 理解WinMain .....	16
1.3.3 完整的WinMain .....	17
1.4 你所学到的 .....	19
1.5 复习测验 .....	19
1.6 自己动手 .....	19
第2章 侦听Windows消息 .....	20
2.1 编写一个真正的Windows程序 .....	20
2.1.1 理解InitInstance .....	23
2.1.2 理解MyRegisterClass .....	25
2.1.3 晒一晒WinProc的秘密 .....	27
2.2 什么是游戏循环 .....	31
2.2.1 老的WinMain .....	31
2.2.2 WinMain和循环 .....	33
2.3 GameLoop项目 .....	35
GameLoop程序的源代码 .....	36

2.4 你所学到的 .....	42
2.5 复习测验 .....	42
2.6 自己动手 .....	43
第3章 初始化Direct3D .....	44
3.1 初识Direct3D .....	44
3.1.1 Direct3D接口 .....	44
3.1.2 创建Direct3D对象 .....	45
3.1.3 让Direct3D转起来 .....	47
3.1.4 全屏模式的Direct3D .....	55
3.2 你所学到的 .....	56
3.3 复习测验 .....	56
3.4 自己动手 .....	57

## 第二部分 游戏编程工具箱

第4章 绘制位图 .....	60
4.1 表面和位图 .....	60
4.1.1 主表面 .....	61
4.1.2 从离屏(off-screen)表面 .....	62
4.1.3 Create_Surface示例 .....	64
4.1.4 从磁盘装载位图 .....	68
4.1.5 Load_Bitmap程序 .....	69
4.1.6 代码再利用 .....	73
4.2 你所学到的 .....	73
4.3 复习测验 .....	73
4.4 自己动手 .....	73
第5章 从键盘、鼠标和控制器 获得输入 .....	74
5.1 键盘输入 .....	74
5.1.1 DirectInput对象和设备 .....	74
5.1.2 初始化键盘 .....	75
5.1.3 读取键盘按键 .....	77
5.2 鼠标输入 .....	77

5.2.1 初始化鼠标 .....	77	7.2 你所学到的 .....	139
5.2.2 读取鼠标 .....	78	7.3 复习测验 .....	140
5.3 Xbox 360控制器输入 .....	79	7.4 自己动手 .....	140
5.3.1 初始化XInput .....	80	第8章 检测精灵碰撞 .....	141
5.3.2 读取控制器状态 .....	81	8.1 边界框碰撞检测 .....	141
5.3.3 控制器振动 .....	82	8.1.1 处理矩形 .....	141
5.3.4 测试XInput .....	82	8.1.2 编写碰撞函数 .....	142
5.4 精灵编程简介 .....	88	8.1.3 新的精灵结构 .....	143
5.4.1 一个有用的精灵结构 .....	90	8.1.4 为精灵的缩放进行调整 .....	144
5.4.2 装载精灵图像 .....	91	8.1.5 边界框演示程序 .....	144
5.4.3 绘制精灵图像 .....	91	8.2 基于距离的碰撞检测 .....	148
5.5 Bomb Catcher游戏 .....	92	8.2.1 计算距离 .....	149
5.5.1 MyWindows.cpp .....	93	8.2.2 编写计算距离的代码 .....	149
5.5.2 MyDirectX.h .....	95	8.2.3 测试基于距离的碰撞 .....	150
5.5.3 MyDirectX.cpp .....	97	8.3 你所学到的 .....	151
5.5.4 MyGame.cpp .....	103	8.4 复习测验 .....	151
5.6 你所学到的 .....	107	8.5 自己动手 .....	151
5.7 复习测验 .....	107	第9章 打印文本 .....	153
5.8 自己动手 .....	108	9.1 创建字体 .....	153
第6章 绘制精灵并显示精灵动画 .....	109	9.1.1 字体描述符 .....	153
6.1 什么是精灵 .....	109	9.1.2 创建字体对象 .....	154
6.2 装载精灵图像 .....	109	9.1.3 可重用的MakeFont函数 .....	154
6.3 透明的精灵 .....	111	9.2 使用ID3DXFont打印文本 .....	155
6.3.1 初始化精灵渲染器 .....	112	9.2.1 使用DrawText打印 .....	155
6.3.2 绘制透明的精灵 .....	113	9.2.2 文本折行 .....	156
6.4 绘制动画的精灵 .....	120	9.3 测试字体输出 .....	156
6.4.1 使用精灵表 .....	120	9.4 你所学到的 .....	159
6.4.2 精灵动画演示 .....	123	9.5 复习测验 .....	160
6.5 你所学到的 .....	126	9.6 自己动手 .....	160
6.6 复习测验 .....	126	第10章 卷动背景 .....	161
6.7 自己动手 .....	126	10.1 卷动 .....	161
第7章 精灵变换 .....	127	10.1.1 背景和布景 .....	162
7.1 精灵旋转和缩放 .....	127	10.1.2 从图片单元创建背景 .....	162
7.1.1 2D变换 .....	129	10.1.3 基于图片单元的卷动 .....	163
7.1.2 绘制变换了的精灵 .....	132	10.1.4 基于图片单元的卷动项目 .....	163
7.1.3 Rotate_Scale_Demo程序 .....	134	10.2 动态渲染图片单元 .....	168
7.1.4 带有变换的动画 .....	136	10.2.1 图片单元地图 .....	169



10.2.2	使用Mappy创建图片单元地图 .....	170
10.2.3	Tile_Dynamic_Scroll项目 .....	174
	Tile_Dynamic_Scroll源代码 .....	175
10.3	基于位图的卷动 .....	180
10.3.1	基于位图的卷动理论 .....	180
10.3.2	位图卷动演示 .....	181
10.4	你所学到的 .....	184
10.5	复习测验 .....	184
10.6	自己动手 .....	184
第11章	播放音频 .....	186
11.1	使用DirectSound .....	186
11.1.1	初始化DirectSound .....	187
11.1.2	创建声音缓冲区 .....	187
11.1.3	装载波形文件 .....	188
11.1.4	播放声音 .....	188
11.2	测试DirectSound .....	189
11.2.1	创建项目 .....	189
11.2.2	修改MyDirectX文件 .....	191
11.2.3	修改MyGame.cpp .....	193
11.3	你所学到的 .....	199
11.4	复习测验 .....	199
11.5	自己动手 .....	199
第12章	3D渲染基础 .....	200
12.1	3D编程介绍 .....	200
12.1.1	3D编程的关键组成部分 .....	200
12.1.2	3D场景 .....	201
12.1.3	转移到第三维 .....	204
12.1.4	掌握3D管线 .....	205
12.1.5	顶点缓冲区 .....	206
12.1.6	渲染顶点缓冲区 .....	208
12.1.7	创建四边形 .....	209
12.2	带纹理的立方体示例 .....	211
	MyGame.cpp .....	213
12.3	你所学到的 .....	219
12.4	复习测验 .....	219
12.5	自己动手 .....	220
第13章	渲染3D模型文件 .....	221
13.1	创建及渲染后援网格 .....	221
13.1.1	创建后援网格 .....	221
13.1.2	渲染后援网格 .....	223
13.1.3	Stock_Mesh程序 .....	224
13.2	装载并渲染模型文件 .....	226
13.2.1	装载.X文件 .....	226
13.2.2	渲染完整的模型 .....	231
13.2.3	从内存中删除一个模型 .....	231
13.2.4	Render_Mesh程序 .....	232
13.3	你所学到的 .....	239
13.4	复习测验 .....	239
13.5	自己动手 .....	240
 <b>第三部分 游戏项目</b> 		
第14章	Anti-Virus(反病毒)游戏 .....	242
14.1	Anti-Virus游戏 .....	242
14.1.1	游戏玩法 .....	243
14.1.2	游戏源代码 .....	251
14.2	你所学到的 .....	264
14.3	复习测验 .....	264
14.4	自己动手 .....	264
 <b>第四部分 附 录</b> 		
附录A	配置Visual C++ .....	268
附录B	可进一步学习的资源 .....	274
附录C	各章测验答案 .....	278
附录D	附加示例 .....	287

### 第三部分 游戏项目

第14章	Anti-Virus (反病毒) 游戏	242
14.1	Anti-Virus游戏	242
14.1.1	游戏玩法	243
14.1.2	游戏源代码	251
14.2	你所学到的	264
14.3	复习测验	264
14.4	自己动手	264

## 第四部分 附 录

附录A	配置Visual C++ .....	268
附录B	可进一步学习的资源 .....	274
附录C	各章测验答案 .....	278
附录D	附加示例 .....	287

## 第一部分 >>>

# Windows 和 DirectX 游戏编程引言

---

本书第一部分介绍 Windows API（应用程序编程接口），这是开始 DirectX 编程之前必须掌握的基础知识。前两章通过讲解编写简单 Windows 程序的方法、Windows 消息系统的工作原理以及创建不停息的消息循环（正是它给 Windows 程序带来了高性能）的方法，让读者对 Windows 的工作原理有一个大概的了解。第 3 章介绍 DirectX，读者在这里将学习创建 Direct3D 渲染设备以及设置渲染系统的方法。

- 第 1 章 Windows 初步
- 第 2 章 倾听 Windows 消息
- 第 3 章 初始化 Direct3D



# 第 1 章 Windows 初步

让人趋之若鹜、不掌握不痛快的计算机编程技术中，游戏编程显然是最复杂的编程形式之一。游戏不仅是巨大的技术成就，更是一种艺术工作。许多在技术上令人惊奇的游戏无人问津，而在技术上不是那么精湛的游戏却大行其道，给制作者带来滚滚财源。尽管我们的最终目标是要成为一名游戏程序员，但作为爱好，这会是你所有的爱好中最让人享受的一种，它所带来的感受可谓有苦有甜。而我希望你已经做好了进行这一冒险的准备！本章提供的是开始编写 Windows 游戏所需的重要信息，它是后面两章的前导，在那里会给出 Windows 程序机制的概要介绍。

本章将展示一个简单的 Windows 程序。这些信息对后续三章的学习很重要，因为需要依靠这些知识将读者带入 DirectX 的世界。如果对这些介绍性的知识掌握不牢，那么以后还需要随时回来复习，因为随后的章节将依赖于我们对 Windows 工作原理的基本理解。如果已经有编写 Windows 程序的经验，则将对学习本书非常有帮助，不过，我并不做这种假定，而是在此给出 Windows 程序的基础知识，这些正是开始编写 DirectX 代码所需的。

实际上，只要投入其中，就不难发现 Windows 编程其实颇为有趣！虽然有些代码看起来可能像外星文字，但很快你就会非常熟悉它们。如果本章的内容让你觉得不知所措，那么不要太过担心，因为后续章节中还会有重复，这样会让你牢记要点所在。本章的目标是展示编写一个简单的 Windows 程序、创建项目、键入代码及编译、运行这个程序的方法。

本章将学到：

- 如何正确看待游戏编程。
- 如何按需选择最好的编译器。
- 如何创建 Win32 应用程序项目。
- 如何编写简单的 Windows 程序。

## 1.1 Windows 编程概述

如果你是 Windows 编程新手，那么将体验到很棒的经历！因为对于编写游戏，Windows 是一个重要的操作系统（不过以前可不总是如此）。首先，Windows 下有许多很棒的编译器和语言。其次，它是世界上最流行的操作系统，任何针对 Windows 写的游戏都有流行起来的潜力。而 Windows 的第三个伟大之处在于有令人惊异的 DirectX SDK 为我们效劳。DirectX 不仅是如今最广泛使用的游戏编程库，它还很容易上手。不过不要误解我的意思——Direct X 易于学习，但想精通它却是另一回事。本书将教授读者如何使用它，或者说如何运用它来创建我们自己的游戏。如果要精通它，则单单这本书所能提供的还远远不够。DirectX 非常值得花时间去学习，尤其是如果你想跟上游戏开发的最新研究成果的话（因为如今大多数关于游戏开发的文章和书籍的着重点都是 DirectX）。

在开始编写 DirectX 代码之前，需要学习如何编写简单的 Windows 应用程序，并且学习

Windows 处理消息的方法。那么，就让我们从头开始吧！什么是 Windows？

Windows 是一个多任务、多线程的操作系统。这句话的意思是，Windows 可以同时运行多个程序，而这些程序中的每一个又都可以有许多运行中的线程。不难想象，这样的操作系统架构能够和诸如 Intel Core2 和 i7 这样的多核处理器良好共事。

**建议** 作为参考，本书中的程序（和截图）是在一台具备 Intel Core2Quad Q6600 处理器、2 GB DDR2 内存、Nvidia 8800GT 512 MB DDR3 视频卡的 PC 上开发的。在编写本书时，这是一台性能处于中上游水平的 PC。

### 1.1.1 认识 Windows

没有几个操作系统能像 Windows 这样一个版本一个版本地逐级延伸。目前使用中的许多 Windows 版本（本书编写时主要是 Windows Vista 和 Windows XP）之间差别不大，为某个版本编写的程序几乎无需修改就可以在另一个版本上运行。例如，在 1998 年使用 Microsoft Visual C++ 6.0 在 Windows NT 4.0 或 Windows 98 下编译的一个程序仍旧可运行于最新版本的 Windows XP 和 Vista 之上。你的游戏库中或许还有几个 20 世纪 90 年代后期出品的支持 DirectX 早期版本的游戏（例如 DirectX 8.0）。如果这样的游戏仍旧可在新 PC 上运行，你也无需惊奇。这无疑是非常好的事情，因为这是一个我们可以依赖多年的、能够运行我们的代码的平台。而这也是 Windows PC 游戏开发人员不甚满意而控制台游戏（console game）开发人员较为满意的地方之一，因为我们可以相信控制台系统不会改变，而 PC 变化太快，在某些情况下，有些游戏在一两年之后就产生技术问题了。

好，我们证实了 Windows 程序的长寿（在软件工业中也称为“保质期”）。那么，Windows 到底能做什么呢？

**建议** 在本书中，凡是提及“Windows”的地方，所指都包括了与当前主题——PC 和游戏编程相关的最新 Windows 版本。也就是说，应该包括所有以前的、当前的和将来的兼容 Windows 版本。从实用的角度来说，实际上这仅限于 32 位程序。从这里开始，凡是提及“Windows”的地方，我们均可认为其包含了所有诸如这样的版本：Windows XP、Windows 2003、Windows Vista 和 Windows 7。例如，本书的前两个版本分别使用 Windows 2000 和 Windows XP 开发的，但它的源代码在最近 5 年内只需很小甚至不需要更改。

根据所编写的程序类型不同，Windows 编程可简可繁。如果你有编写应用程序的经验和开发背景，那么就会对图形用户界面（GUI）编程的复杂性有很好的理解。只需几个菜单和窗体，就足以把你淹没在数十个（就算到不了数百个）控件中难以自拔。作为多任务的操作系统，Windows 非常优秀，因为它是消息驱动的。面向对象编程的拥护者争辩说 Windows 是个面向对象的操作系统。事实上它不是。在工作方式上，当今最新的 Windows 版本几乎和 Windows 的早期版本（例如老的 Windows 286、Windows 3.0 等）一样——驱动操作系统的是消息，不是对象。操作系统就如同人类的神经系统，只是没那么错综复杂。如果将人类的神经系统以抽象的方式来简化，那么就会看到在人体中，刺激通过神经元从感官器官传递到大脑，而后从大

脑传递到肌肉。

**建议** 虽然 64 位计算是将来的潮流，但对于程序员来说，它不会像从 16 位到 32 位的转换那样成为一个大问题，因为处理器、操作系统和开发工具都同时转换了，所以这种转换将几乎不可察觉（现实就是如此）。

### 1.1.2 Windows 消息机制

下面通过一个常见的场景来帮助我们对操作系统和人类神经系统进行比较。假设你的皮肤上的神经检测到了某些事件，例如温度的改变或者某些东西对你的触摸。如果使用右手的手指触摸左手臂，会发生什么？你会“感觉”到触摸。为什么呢？当你触摸你的手臂时，感觉到触摸的不是手臂而是大脑。“触摸”这一感觉并非由手臂本身感受到，其实是大脑定位到了事件，于是你识别出了触摸的来源。这几乎就如同对中枢神经系统中的神经元进行查询，看它们是否参与了“触摸事件”。大脑“看到”触摸消息传递链中的神经元，于是就可确定手臂上发生触摸的位置。现在，请触摸你的手臂，并且在手臂上来回移动手指，你感觉到发生什么了吗？这不是持续的“模拟”测量，因为在皮肤上有许多离散的触摸敏感神经元。运动的感觉实际上是以数字的方式传递到大脑的。你可能会反驳我的论断，认为压力的感觉是模拟的。在这里我们深入到了某种抽象概念中，但笔者认为压力的感觉是以离散增量传递到大脑的，它并不是以电容性的模拟信号来传递的。

这个概念和 Windows 编程有什么联系呢？触摸的感觉与 Windows 消息有非常类似的工作方式。外部的事件，例如鼠标单击，会导致小的电信号从鼠标传递到 USB 口，然后再进入系统总线，可以认为这是计算机的神经系统。操作系统（Windows）从系统总线检出这一信号并且生成一个消息传递给正在运行的应用程序（例如我们的游戏）。而后，程序就如有意识的心灵那样对“触摸的感觉”做出反应。计算机的潜意识（处理所有事件处理逻辑的操作系统）将这一事件“呈现”给程序，让其知晓。

**建议** 随着时间的推移，高级信息系统似乎趋向于模仿神经世界，未来我们最终构造出的终极版的超级计算机，可能会和人类大脑相像。

目前还有一个问题。人类可没有两个大脑。还记得笔者关于技术模仿生物大脑的评述吗？当今大多数处理器厂家都朝着在单个硅芯片上集成多个处理器内核的方向前进。多核系统在今天已是常态，绝不是特例（本书在 2004 年首次出版时还不是这样）。

#### DirectX 9、10 还是 11？

编写本书时 DirectX 10 已经存在两年时间了，而且 DirectX 11 已经处于早期的测试阶段。实际上，在 CD-ROM 中包含的 DirectX SDK 中，就有 DirectX 11 的一个早期版本。这对于我们这些仍旧使用 DirectX 9（这是本书所关注的）编写游戏程序的人来说意味着什么呢？

对 DirectX 9 的支持仍旧如此广泛的原因是它支持 Windows XP，而 Windows XP 仍旧是最广泛使用的 Windows 版本（因为大多数个人和企业还没有升级到 Vista）。DirectX 10 及更新的版本还不能运行于 Windows XP，但 DirectX 9 的代码可运行于任何 Windows 的现代版本上——这就



是 DirectX 9 仍旧流行的原因。

就算 Windows 7 正式发布时会带 DirectX 11，但我们可继续编写 DirectX 9 代码，因为在可预见的将来，DirectX SDK 将继续提供所有编译并运行 DirectX 9、10 和 11 程序所需的头文件、库文件和 DLL 文件。

### 1.1.3 多任务

首先，Windows 是一个抢占式多任务操作系统，也就是说计算机可以同时运行多个程序。Windows 通过让每个程序运行很短的时间——以毫秒，或者秒的千分数计算的时间来实现这一特性。从一个程序非常快地跳转到另一个程序称为时间分片，Windows 通过为内存中的每个程序创建虚拟地址空间（一个小的“模拟的”计算机）来处理时间分片。每当 Windows 跳转到下一个程序时，会储存当前程序的状态，以便在轮到这个程序接受处理器时间时能够接着执行。这些状态包括处理器寄存器值和任何可能被下一个进程覆盖的数据。然后，当程序以时间分片方案重新得到处理时，这些值就会被恢复到处理器寄存器中，这样程序可以回到离开的位置继续执行。

**建议** 不要觉得这样做是对处理器周期的浪费，要知道就在这么几个毫秒中，处理器可运行好几十万条指令。现代处理器已经可达到每秒 10 亿次浮点运算的水平，在短短的“时间片”中很容易就能完成巨量的数学计算。

可以认为，Windows 有其自己的基于事件的中枢神经系统。当我们按下一个键，就会有消息从按键事件中创建并且在系统中散播，直到有程序检出这个消息并使用它。因为这里提及“散播”，所以在这里需要澄清一点。Windows 3.0、3.1 和 3.11 是非抢占式操作系统，从技术上说它们只是 16 位 MS-DOS 之上的非常先进的程序而已。Windows 的这些早期版本更像是 MS-DOS 的外壳，而不是真正的操作系统，所以无法真正地“拥有”整个计算机系统。我们可以编写一个可完全接管系统的 Windows 3.x 程序，无需为其他程序释放处理器周期。只要愿意，甚至可以锁住整个操作系统。早期的 Windows 程序为了获得“Windows Logo”认证（在那个时候是重要的营销问题），必须释放对计算机资源的控制。Windows 95 是第一个 32 位版的 Windows，而且它是这一操作系统家族的革命性进步，因为它是抢占式操作系统。

这里的意思是，操作系统有非常低级的内核用于管理计算机系统，没有任何程序可以接管这一系统，不像 Windows 3.x 那样。抢占式意味着操作系统可以抢占一个程序的运行，使其暂停，并且，操作系统可在以后让程序再次启动运行。当有许多程序和进程（每个都有一个或多个线程）请求处理器时间时，就称其为时间分片系统，这也是 Windows 的工作方式。不难想象，在使用这样的操作系统时，多处理器系统实在是很有优势。

对于游戏开发者而言，多核的 Intel 或 AMD 系统是很棒的配置。首先，SMP（symmetric multiprocessing，对称多处理）处理器通常有更多的内部高速缓冲存储器。处理能力是越大越好！在过去，玩游戏或者开发游戏时可能必须关闭大多数运行中的其他应用程序，但现代的多核系统可轻而易举地处理多应用程序同时运行的问题，在编写游戏时根本觉察不出系统运行有拖拖拉拉的现象。当然，巨量的内存也有帮助——游戏开发至少要 2GB RAM。既然已经开工，那么就购买你的系统上可处理的最快的内存芯片吧，因为对于计算机硬件而言，速度比容量更好！

图 1-1 显示了非抢占式多任务工作原理的概览。注意，每个程序都接受处理器控制，并且之后必须显式释放控制以便计算机系统正确工作。这样的程序还必须小心的是，不能占用太多处理器时间；实质上，非抢占式操作系统程序必须自发共享处理器。

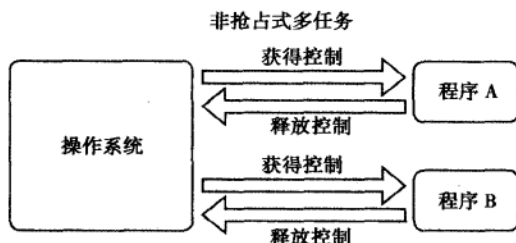


图 1-1 非抢占式多任务要求每个程序自发释放控制。操作系统对应用程序的控制非常有限

图 1-2 显示了抢占式多任务的工作原理。不难看出，图 1-2 与图 1-1 相似（这样易于比较），不过，现在操作系统控制所有的一切，无需等待程序“听话”并且共享处理器时间。在时间片分配的毫秒数满了之后操作系统只需挂起程序即可，在对系统中所有运行中的进程和线程循环一遍之后，再把更多处理器时间交给这个程序。

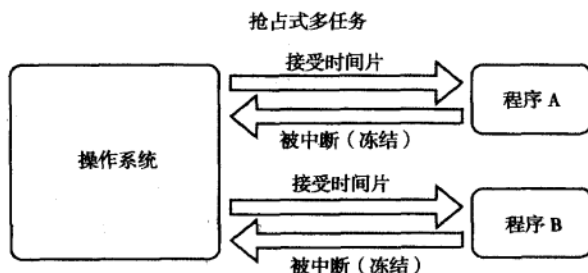


图 1-2 抢占式多任务操作系统对系统有完全的控制，它为每个运行中的进程和线程分配时间片

#### 1.1.4 多线程

**多线程**是将程序分解成多个独立的、为了完成某个任务（或者完成独立的任务）而一起工作的部分。这与系统级别的多任务不是一回事。多线程有点像多-多任务，每个程序都有其自己的运行部分，而这些小程序片段对操作系统执行的时间分片系统毫无察觉。对于主 Windows 程序及其所有的线程，它们对系统有完全的控制，而对操作系统将时间片分配给每个线程或进程并无“知觉”。所以，多线程意味着每个程序能够将处理托付给其自己的迷你程序。例如，象棋程序可以创建一个进程以便在游戏者忙于考虑下一步棋时提前思考。“思考”线程可在等待游戏者的同时继续更新着法和对攻着法。虽然这可以在等待用户输入时使用能够思考的程序循环很容易地实现，但具备将这一过程交付给线程来执行的能力可给程序带来显著的好处。

举个例子来说，我们可以在一个 Windows 程序中创建两个线程并且让每个线程有其自己的循环。对于每个线程，其循环无尽运行并且极快地运行，没有中断。但是在系统级别上，会给每个线程授予一个处理器时间片。依据处理器的速度和操作系统的不同，线程每秒可能会被中断 50、100 甚至 1000 次，但线程对这样的中断毫无察觉（想象一下我们在夜里睡觉时会醒来许多次！与人类不同，计算机并不会注意到！）。图 1-3 说明了程序、进程和线程之间的关系。

**建议** 多线程编程是个迷人的主题，值得我们花时间学习！笔者在《Game Programming All In One》一书的第 3 版（ISBN 1-59863-289-2）中简单讲解了这一主题。讲解了能让多线程编程变成小菜一碟的 Posix Threads 库的使用方法。在读完本书之后，它会是很好的后续材料。笔者发现大多数初学者都能很快地学会 Allegro 游戏库。如果你准备好接受更大的挑战，那么笔者的针对这一主题的新书《Multi-Threaded Game Engine Design》（ISBN 1-4354-5417-0）值得一读。不过，如果对 C++ 和 DirectX 尚不熟练，要做好进行巨量编程训练的准备！

多线程对游戏编程非常有用。在一个游戏循环中涉及的许多任务都可以交付给可独立执行的不同线程实现，每个线程都与主程序通信。其中一个线程可用于自动处理屏幕更新。而后，程序必须做的就是确认所有的对象都在屏幕上，并且双缓存以特定的时间进行更新，而这个线程将按时执行工作——甚至可能使用内置的计时措施来保证无论使用什么处理器游戏都能以统一的速度运行。大多数流行的游戏引擎都是多线程的，也就是说它们天生就支持多处理器。这对于那些为了买多核系统而花费更多的游戏者而言，实在是物有所值。而独立的游戏服务器（经常提供流行的在线游戏以便游戏者运行他们自己的游戏）如果能够支持多处理器就更为理想，因为它需要大量的处理能力来处理有许多游戏者的大型游戏。双处理器游戏服务器更能够处理大量游戏者。

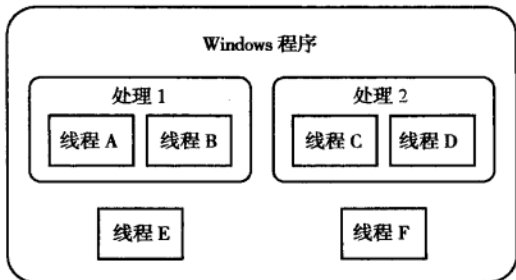


图 1-3 多线程程序可以有多线程处理也可以有独立的线程

**建议** 双缓存是内存中的一种位图映像，可以用它来为游戏绘制图像，然后这一映像会被复制到屏幕上，以非常平滑的渲染效果显示出来。

### 1.1.5 事件处理

到这里，有人可能会问：“Windows 如何与这么多同时运行的程序保持联系呢？”首先，Windows 处理这一问题要求程序必须是事件驱动的。其次，Windows 使用全系统范围内的消息来通信。Windows 消息是操作系统发送给每个运行中的程序的小数据包，它有三个主要内容——窗口句柄、实例标识符和消息类型，用于告诉程序有事件发生。事件通常涉及用户输入，例如鼠标单击或按键，不过事件也可以来自网络库的通信端口或 TCP/IP 套接字（用在多人游戏中）。

每个 Windows 程序必须在消息处理器中检查每一条收到的消息，确定该消息是否适用于本程序。不被识别的消息会向前发送给默认的消息处理器，它会将这些消息送回 Windows 消息流。可将消息想象成鱼——当我们抓到太小的或不喜欢的鱼时，就会把它扔回水里；但我们会把想要的鱼留下来。这与 Windows 事件驱动架构类似：如果程序识别出了一条想要保留的消息，那么程序就会从消息流中取出这条消息，其他程序就看不到它了。

一旦熟悉了 Windows 编程并且学习了处理某些 Windows 消息的方法，就能理解 Windows 消息机制是如何为应用程序（而不仅是游戏）设计的。技巧就是学着“进入”Windows 的消息系统并且将我们自己的代码注入其中，例如 Direct3D 初始化例程或者进行屏幕刷新的函数调用。游戏中的所有动作都通过 Windows 消息系统来处理，拦截并处理与游戏有关的消息是我们的工作。第 2 章将学习如何编写 Windows 程序，在后面两章中将会学到更多关于 Windows 消息的知识。

## 1.2 DirectX 快速概览

为了能够将 DirectX 介绍给大家，在很短的时间内讲解了许多关于 Windows 理论的内容。我们可能对 DirectX 已经如雷贯耳，因为这是一个业内人士口中的时髦语，不过能真正理解它的人不多。我们不是要开始编写 DirectX 代码，而是要先了解它与 Windows 如何协同工作。DirectX 提供 Windows PC 的低级硬件接口，为不使用 Windows API 或 GDI 的游戏提供一致和可靠的函数集（这意味着 DirectX 要快得多）。图 1-4 显示了 DirectX 如何与 Windows API 协同工作。

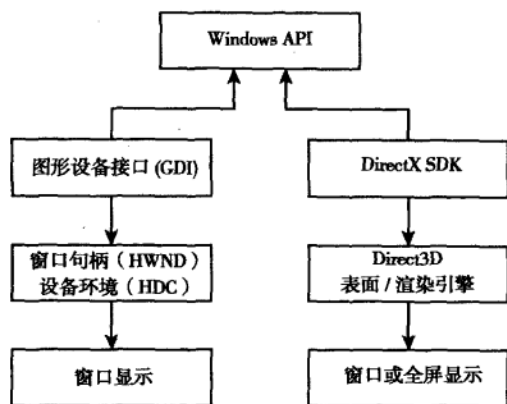


图 1-4 DirectX 9 与 Windows API 协同工作

DirectX 与 Windows 紧密集成，它不能在任何其他操作系统中工作，因为它的功能依赖于 Windows API 中的基础库。以下列出的是 DirectX 组件的纲要信息：

- Direct3D 是对视频卡和图形处理器（GPU）提供访问来渲染 2D 和 3D 图形的渲染组件。这是最重要的组件。
- DirectSound 是以前用于播放波形文件中的数字采样内容的过时的音频组件，它使用多通道音频混合器。为了保持向后兼容，它仍旧包括其中。
- XACT 是补充 DirectSound 的新音频系统，它也用于 XNA Game Studio SDK 中的音频处理。
- DirectInput 是设备输入组件，用于访问诸如键盘、鼠标和游戏杆这样的外设，也支持像飞行控制杆和方向盘这样的特殊硬件。
- XInput 是新的用于访问连接到 Windows PC 上的 Xbox 360 控制器的组件。虽然大多数 Windows 用户没有插接这些控制器，但对于 XNA Game Studio 程序员来说它非常流行。
- DirectPlay 是先前的联网组件，已经不再支持，但为了向后兼容仍旧包括其中。

**建议** XNA Game Studio 是 Visual Studio 的一个 SDK 和插件，其使 Visual Studio 既可以开发用于 Windows 也可以开发用于 Xbox 360 控制台的游戏程序。本书编写时的最新版本是 3.1。XNA 开发人员可在 XNA Creator's Club 网站上发表他们的游戏作品，并且可以在 Xbox Live Arcade 上销售。

## Direct3D 是什么

由于 Direct3D 是 DirectX 最重要的组件，因此这里特别讨论下它。Direct3D 处理游戏中所有的 2D 和 3D 渲染工作。我们将从第 3 章开始学习 Direct3D 的用法。在后面的章节中，将学习如何将位图文件装载到内存中作为纹理，然后绘制该纹理（在 2D 模式中），以及在渲染 3D 对象（称为网格——mesh）时应用纹理。

必须承认，笔者是 2D 游戏的铁杆粉丝，尤其是回合制的战略游戏和经典的闯关游戏，例如古老的“Sid Meier's Civilization”系列（虽然其最新的实现——Xbox 360 和任天堂 Wii 上的 Civilization Revolution 已经完全以 3D 来实现了）。我们仍旧可以使用 Direct3D 来编写 2D 游戏，或使用 2D 位图和精灵来增强 3D 游戏。我们会需要在屏幕上用 2D 字体来显示信息，所以必须学习绘制 2D 图形的方法。就眼前来讲，对 2D 纹理和精灵的简单概述可帮助我们在以后探究 3D 编程时理解 Direct3D。作为一个例子，Civilization Revolution 的任天堂 DS 版就是个 2D 游戏。

为了不至于偏离主题，再重述一遍这一重要的目标：形成对 2D 和 3D 图形的理解并具备创建游戏所需的知识。目的不是让你成为专业的游戏程序员，而是给你足够的信息（和激情！），以便可以自己进入更高层次，学习更多关于这一主题的知识。最终将进入 3D 模型、纹理、光照和编写简单 3D 游戏所需的所有其他主题，但我们将继续使用守旧派的固定功能管线（fixed function pipeline），也就是说我们将不使用任何顶点或像素着色器（shader）（运行于 GPU 之上而不是 CPU 之上的程序）来实现 3D 渲染。我的意思是，希望大家能够从这些资料中获得乐趣，而不是陷入细节之中！因为 3D 游戏编程的细节非常巨大而复杂，通常初学者在听说顶点缓冲区和纹理坐标这样的东西时会目光呆滞的。这么说是因为我知道作为刚刚起步的你，要积累这么多的细节信息，尚需时日。

虽然现在就开始编写 Direct3D 代码也是可能的，但我们迟早都得学习 Windows 编程的基础知识并且熟悉 WinMain 和其他 Windows 内核函数，因为这是 DirectX 程序的核心。

## 1.3 Windows 程序基础

是否准备好编写 Windows 程序了？很好！现在我们将学习使用 C++ 编写一个真正的 Windows 程序。每个 Windows 程序至少包括一个名为 WinMain 的函数。大多数 Windows 程序还包括名为 WinProc 的消息处理器函数用于接收消息（例如按键和鼠标移动）。如果你正在编写一个真正意义上的 Windows 应用程序（例如 3ds Max 或 Microsoft Word 这样的商业软件），那么程序中会有一个非常大而复杂的 WinProc 函数用于处理许多程序状态和事件。不过在 DirectX 程序中，我们无需淹没在事件中，因为我们的主要兴趣在于 DirectX 接口，它们提供了自己的函数来处理事件。DirectX 主要是一个轮询的 SDK，也就是说你必须请求数据，而不是由它将数据扔给



我们（这是 WinProc 的方式）。例如，在开始学习 DirectInput 时，会发现键盘和鼠标输入主要是通过调用函数来检查是否有值更改的方式来收集的。

### 1.3.1 创建第一个 Win32 项目

每个新项目都将是相似的，一旦学会了在 Visual C++ 中创建新项目，就可以使用同样的策略来创建书中所有的其他项目。你可能会问，什么是项目呢？项目实际上是一个管理程序中所有源代码文件的文件。本书中所有的简单程序将只有一个源代码文件（至少在我们构建游戏框架之前），但大多数真正的游戏会有许多源代码文件，例如 Direct3D 例程、DirectInput 代码和 XACT 代码等都会有各自的源代码文件，而且游戏本身还会有主代码。项目记录所有这些源代码文件，并且由编译器的 IDE 进行管理。为了简单起见，本书将使用“Visual C++”这个通用术语，而不使用“Visual Studio 2005 或 2008 或以后的版本”。

**建议** 重要！如果你是 Visual C++ 的新手，而且不知道如何下载及安装这一软件（更不用说如何创建新项目）那么请参考附录 A，其中有完整的如何设置编译器的说明。编写这一附录是为了减少每章中重复的信息。本书将要求读者参考附录 A，而不是每次都讲解创建项目的方法。

让我们在 Visual C++ 中创建一个新项目。打开 File 菜单，选择 New → Project，如图 1-5 所示。这里显示的是 Visual C++ 2008 版，它是本书编写时的最新版本。

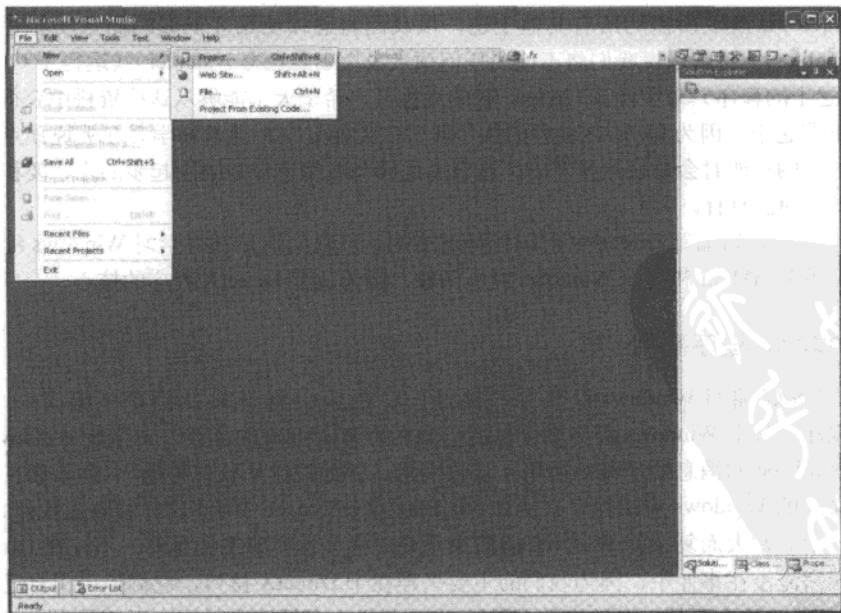


图 1-5 Visual C++ 中的打开新项目对话框

打开左边的 Visual C++ 列表条目，然后选择 Win32 项目模板，如图 1-6 所示。不要迷失在项目模板的列表中，务必使用 Win32 类型，避免混淆。

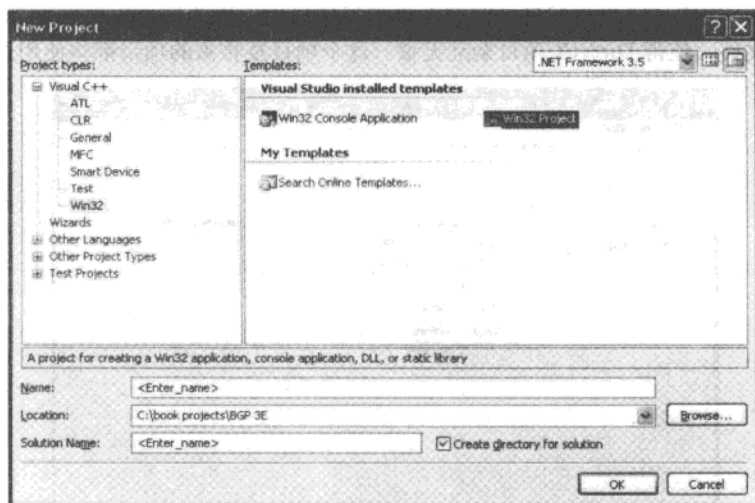


图 1-6 创建新的 Win32 项目

下一步，在 New Project 对话框中，需要在接近底部位置的 Name 字段中输入新项目的名称。在图 1-7 所示的对话框中输入了 HelloWorld 这一名称。请照此给项目命名，然后按 OK 按钮继续。

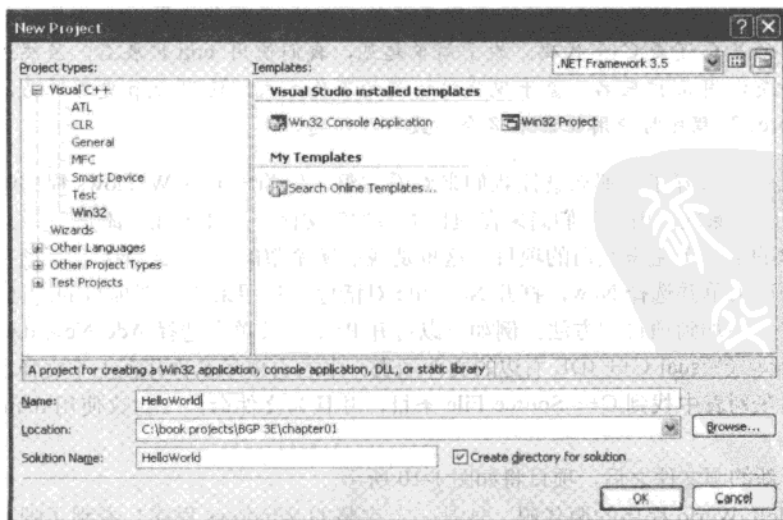


图 1-7 输入新项目的项目名称

接下来是 Application Wizard 对话框。单击左边的 Application Settings 选项卡，打开如图 1-8 所示的对话框。注意，在可选项目中已选择了 Windows application 和 Empty project。一般总是创建“空白项目”，这样可以将我们自己的文件添加到项目中，因为应用程序向导会默认生成太多我们不需要的文件，使得 DirectX 项目杂乱无章。这个项目及将来的项目都将使用这一标准。

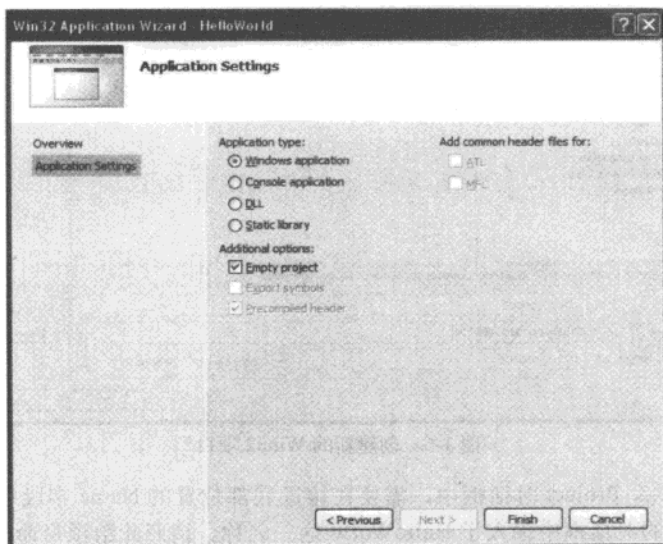


图 1-8 使用 Application Wizard 为新项目选择设置

**建议** 不要被文件扩展名弄糊涂。所有现代的 C++ 编译器都使用 .cpp 文件扩展名，无论编写的是 C 还是 C++ 代码。为了简单起见，我们使用 .cpp 扩展名，虽然过去几年的趋势是使用 .c 扩展名。鉴于现代编译器的工作方式，使用 .cpp 更为容易，因为在编译 DirectX 程序时使用 .c 扩展名会导致一些问题。

新项目已经准备好了，那么就让我们来看看完整（但简单）的 Windows 程序吧，这样能更好地理解它的工作原理。由于我们尚未在项目中添加新文件，因此现在就添加一个。

如果新建的是一个完全空白的项目（这也是我们所希望的），那么就需要给项目添加一个源文件。打开 File 菜单并选择 New，打开 New file 对话框（和用来创建新项目的对话框一样）。有许多将新源文件添加到项目的方法。例如可以打开 Project 菜单并选择 Add New Item，也可以在 Solution Explorer（Visual C++ IDE 右边的文件列表）中右击项目名称并在上下文弹出菜单中选择同一个选项。在列表中找到 C++ Source File 条目，并且为文件命名（建议使用 main.cpp），如图 1-9 所示。

在添加了新的源文件之后，项目将如图 1-10 所示。

以下是 HelloWorld 程序的源代码。这是一个完整的 Windows 程序！看到了吧，如果剔除所有那些应用程序的东西，例如编写游戏时并不需要的菜单，那么 Windows 编程并不真的那么困

难。当然，这是一个过分简单化的示例，因为这个程序只做一件事情：显示一个消息框，然后就退出。

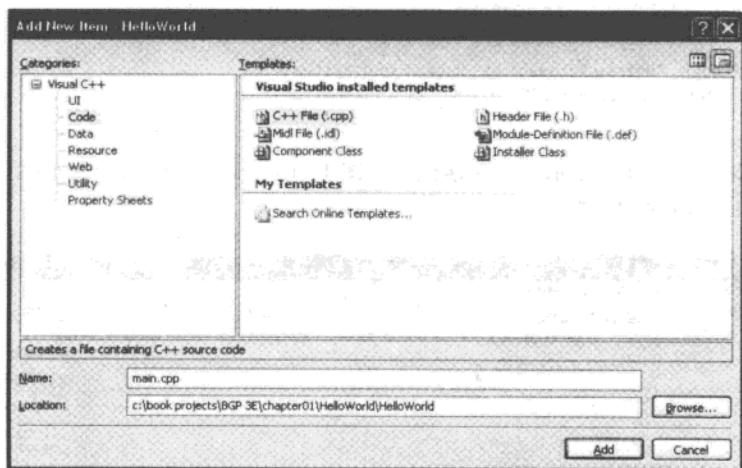


图 1-9 在空白项目中添加一个新文件——main.cpp

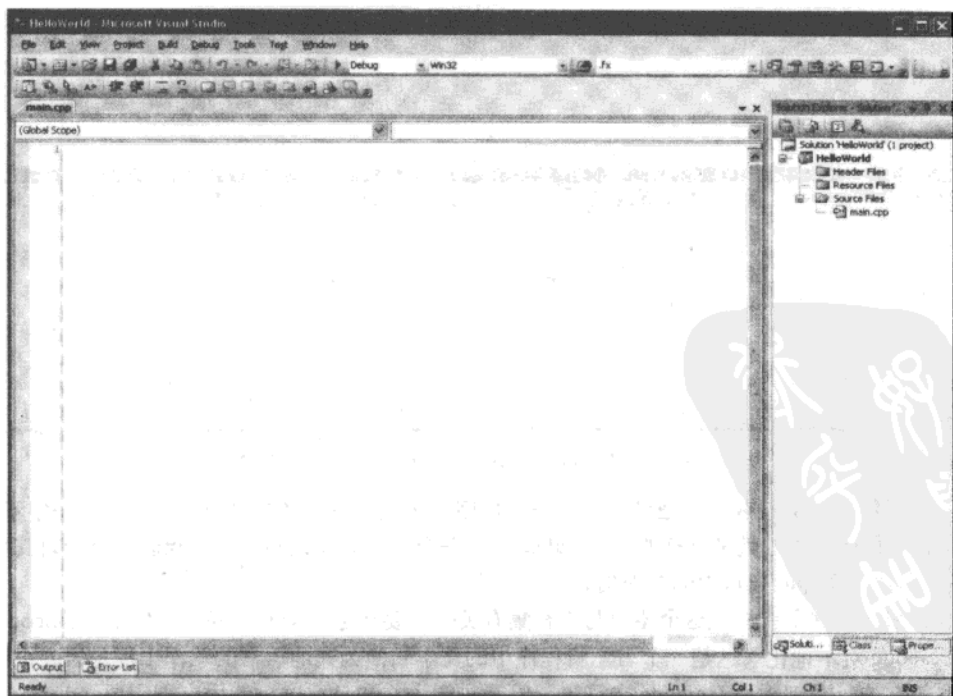


图 1-10 新的 main.cpp 源文件已添加到了项目中并等待用户输入源代码

```
// Beginning Game Programming, 2nd Edition
// Chapter 1 - HelloWorld program
#include <windows.h>
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nShowCmd)
{
    MessageBox(NULL, "Motoko Kusanagi has hacked your system!",
        "Public Security Section 9", MB_OK | MB_ICONEXCLAMATION);
}
```

这个程序仅仅在屏幕上显示一个对话框，如图 1-11 所示。接下来编译并运行这个程序，查看它的运行效果。现在按 F5 键。

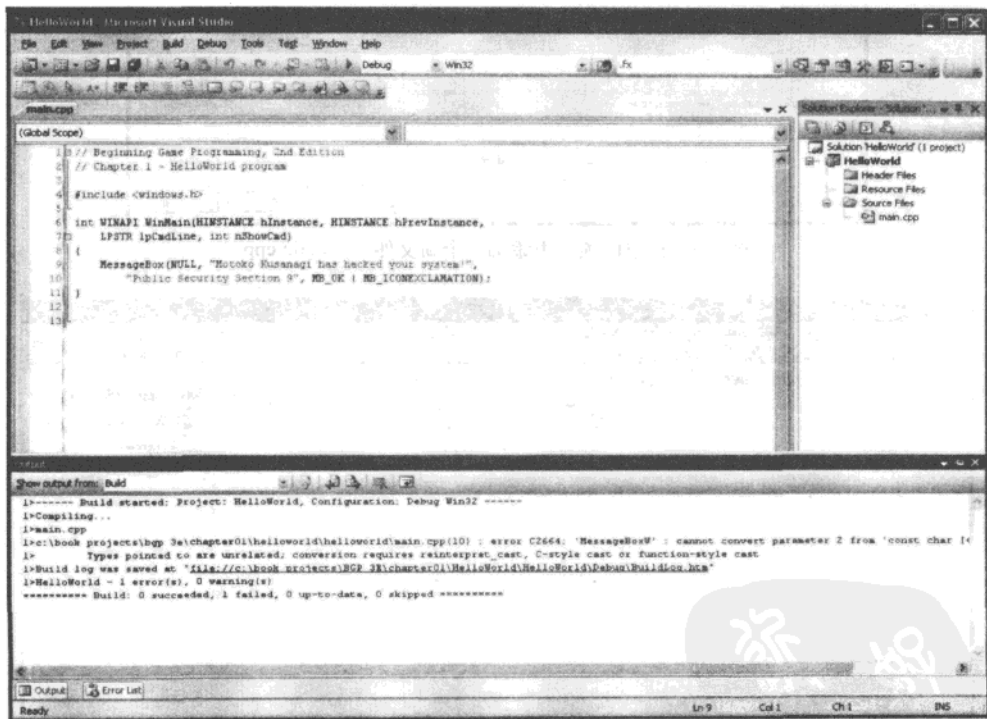


图 1-11 在 Output 窗口中显示的 HelloWorld 程序的编译器错误

发生了什么事情？哦，真是糟糕，第一个程序居然就导致编译器错误，尽管我们一字不漏地键入了整个程序。不过就是这样的。Visual C++ 是一个又大又复杂的软件，我们需要学习其用法。图 1-11 显示了 Output 窗口中的错误信息。

现在来解决这个错误。这个错误与字符集有关——要么是 ANSI（8 位），要么是 Unicode（16 位）。Unicode 对本地化（这是一个表示将程序中的文本转换为其他语言的软件工程术语）很重要。并不是所有的语言都需要 Unicode 字符集，但某些像中文和日文这样的语言则需要。我们本应使



用并非 C++ 标准的那些时髦代码（例如著名的“L”字符和 TCHAR）将所有的代码都写成支持 Unicode 字符串的形式，但我们想编写遵循标准的软件，这就意味着要尽可能避免 Microsoft 式的时髦。为什么呢？我们并不是要在 Windows 以外的其他任何操作系统中编译代码，所以是否使用 Unicode 有什么不同呢？虽然 DirectX 代码只能运行于 Windows 系统中，但我们并不总是使用 Visual C++ 来编译代码！还有其他编译器！笔者在许多场合使用过 Turbo C++ 和 Dev-C++ 来编译 DirectX 代码，并且尽可能让代码既灵活又多用。也就是说，不把代码限制在任何特殊的变通方案中。所以，我们将项目转换成 ANSI 并忽略全部关于 Unicode 的问题。打开 Project 菜单并选择 HelloWorld Properties 选项，打开 Project Properties 对话框，如图 1-12 所示。

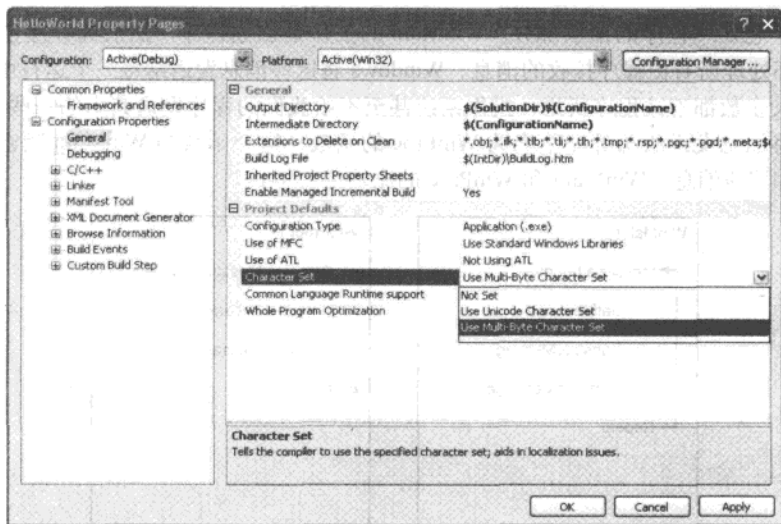


图 1-12 更改项目所使用的默认字符集以避免出现 Unicode 错误

在关闭该对话框之后，按 F5 键再次运行程序。这次弹出了一个消息框，如图 1-13 所示。从这个示例中我们学到的最重要的东西是什么呢？那就是，WinMain 并不需要是一个又大又复杂的应用代码大杂烩。

**建议** 在使用 Visual C++ 编译程序时，可执行文件位于名为 Debug 的文件夹中（在项目文件夹内）。

在了解了非常简单的 Windows 程序到底是什么样之后，下面将更进一步地走进 Windows 编程的神秘世界，学习创建一个真正的窗口并在上面画些东西。使用 MessageBox 总是有点投机取巧的感觉！

我们需要的是一个自己的窗口，接下来将创建它。按照攀爬学习曲线的传统，我们将在下一个项目中对这个小示例

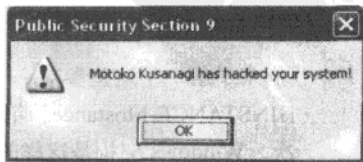


图 1-13 HelloWorld 程序的输出

做一些扩展，展示创建标准程序窗口并在上面绘图的方法。这是准备好使用 Direct3D 之前的又一步骤。

### 1.3.2 理解 WinMain

刚刚学过，每个 Windows 程序都有一个名为 WinMain 的函数。WinMain 是控制台 C++ 程序中 main 函数的 Windows 等同体，并且是 Windows 程序的初始进入点。虽然程序中最重要函数将是 WinMain，但在创立好了消息调用之后，我们就可能不再回到 WinMain，而是在程序的另一部分工作了。

从 16 位的 Windows 3.1 开始 WinMain 就没怎么变过，那还是在 1992 年呢！WinMain 是老板、工头，它处理程序的顶层部分。WinMain 的工作是创立程序，然后为程序创立主消息循环。这个循环处理所有由程序接收的消息。Windows 将这些消息发送给每一个运行中的程序。这些消息中的大多数都不被程序所用，操作系统甚至不给我们的程序发送其中的一些消息。通常，WinMain 会将消息发送给另外一个名为 WinProc 的函数，这个函数与 WinMain 紧密协作，处理用户的输入和其他消息。WinMain 和 WinProc 的比较见图 1-14。

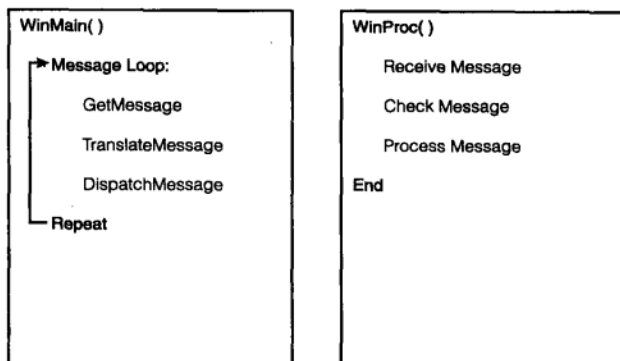


图 1-14 WinMain 和 WinProc 协同工作以处理应用程序事件（例如绘制屏幕及响应鼠标单击）

#### WinMain 函数调用

WinMain 的函数调用如下：

```
int WINAPI WinMain( HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPTSTR lpCmdLine,
                   int nCmdShow )
```

我们来了解一下这些参数：

- HINSTANCE hInstance。第一个参数标识被调用的程序的实例，因为一个程序可运行许多次。Windows 架构是这样的：为了保护内存，程序代码实际运行于一个单一的内存空间中，而程序数据和变量储存于各自的内存空间中。hInstance 参数告诉程序要运行的是哪个实例。

我们会在第一个实例中初始化程序（后面会介绍）。但如果程序要在 Windows 中运行多次，则通常的做法是简单地停止新实例的运行（也在后面介绍）。

- **HINSTANCE hPrevInstance**。第二个参数标识程序的前一个实例，而且与第一个参数有关。如果 hPrevInstance 为 NULL，那么这就是程序的第一个实例。在初始化当前实例之前，可检查 hPrevInstance 的值。这对游戏编程至关重要！我们绝不会希望我们的游戏在同一时间有两个实例在运行。
- **LPTSTR lpCmdLine**。第三个参数是包含传递给程序的命令行参数的字符串，用于告诉程序使用某些选项，例如可使用 debug 标识将程序的执行转储到文本文件中。通常 Windows 程序使用设置（INI）文件保存运行时的程序参数。不过在许多时候我们将使用程序参数，例如一个图像查看程序经常传递将要显示的图片文件名。
- **int nCmdShow**。最后一个参数指定程序窗口的显示方式。

不难注意到，WinMain 返回一个值，因为在函数调用的前面有 int WINAPI 的字样。这也是标准做法，关于它可追溯到 Windows 3.x 的时代。返回值为零表示程序没有进入主循环并提前终止。任何非零值都表示成功。

### 1.3.3 完整的 WinMain

以下所列的是在应用程序代码中经常使用的更标准的 WinMain 版本。在代码清单之后将对这个函数的各个部分进行讲解。

```
int WINAPI WinMain(HINSTANCE hInstance,
                  HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine,
                  int nCmdShow)
{
    // declare variables
    MSG msg;

    // register the class
    MyRegisterClass(hInstance);

    // initialize application
    if (!InitInstance(hInstance, nCmdShow)) return FALSE;

    // main message loop
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
```

考虑到 WinMain 函数要为程序处理 Windows 消息，这个函数再简单也不会超过上面这段代码（后面会讲解这些新东西！）。即使最简单的图形程序也需要处理消息。信不信由你，即使是实现诸如将“Hello World”打印到屏幕上这么简单的事情也需要等待一个用于绘制屏幕的消息的

到来。怒了吧？如果习惯于通过调用函数来取得想要的结果（例如在屏幕上显示文本），那么对于消息处理的确需要花些代价来适应。幸运的是，我们不会在 Windows 基础上花费太多时间，因为我们很快就会进入 DirectX 的王国。一旦初始化了 DirectX3D 并且让游戏循环运行，通常就无需再回过头来编辑 WinMain 中的任何代码了。

现在，讲解 WinMain 内所发生的事情。由于大家已经对这个函数调用有所熟悉，因此我们可直接进入真实代码中。第一段声明 WinMain 中要使用的变量：

```
// declare variables
MSG msg;
```

MSG 变量以后由 GetMessage 函数使用，用于取得每个 Windows 消息的详细信息。接下来，程序由以下代码来初始化：

```
// register the class
MyRegisterClass(hInstance);

// initialize application
if (!InitInstance (hInstance, nCmdShow))
    return FALSE;
```

这段代码使用了由 Windows 传递给 WinMain 的 hInstance 变量。这个变量而后被传递给 InitInstance 函数。InitInstance 位于程序的更下部，它完成检查程序是否已经在运行中，然后创建主程序窗口的基本工作。下面很快会介绍 MyRegisterClass 函数。最后，我们看一看在程序中用于处理所有消息的主循环：

```
// main message loop
while (GetMessage(&msg, NULL, 0, 0))
{
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
```

WinMain 这部分中的 while 循环会持续运行到永远，除非有停止程序的消息到来。GetMessage 函数调用如下所示：

```
BOOL GetMessage( LPMSG lpMsg,
                 HWND hWnd,
                 UINT wMsgFilterMin,
                 UINT wMsgFilterMax )
```

我们来认识一下参数：

- LPMSG lpMsg。这个参数是一个指向用于处理消息信息的 MSG 结构的长指针。
- HWND hWnd。第二个参数是特定窗口消息的句柄。如果传递的是 NULL，则 GetMessage 将为当前程序实例返回所有消息。
- UINT wMsgFilterMin 和 UINT wMsgFilterMax。这两个参数告诉 GetMessage 返回一定范围内的消息。在整个 Windows 程序中对 GetMessage 的调用是最具决定性的一行代码！在 WinMain 中要是没有这一行内容，程序就失去知觉，无法对世界做出响应了。

GetMessage 循环中的两行核心代码处理 GetMessage 返回的消息。Windows API 参考文档指

出, TranslateMessage 函数用于将虚拟键盘消息翻译成字符消息, 然后通过 DispatchMessage 发送回 Windows 消息系统中。这两个函数将一起为游戏窗口创立要在 WinProc (窗口回调函数) 中接收的消息, 例如 WM\_CREATE 用于创建一个窗口, 而 WM\_PAINT 用于绘制窗口。下一章将讲解 WinProc。如果你感觉 Windows 消息机制难以理解, 不要担心, 因为这只是与 DirectX 一起工作的一个初期形式, 一旦编写了 Windows 消息循环, 就无需再与它打交道, 而是可以专注于 DirectX 代码了。现在让我们稍事休息, 然后在下一章中继续学习 WinMain。

## 1.4 你所学到的

本章是为 DirectX 编码做准备, 学习了 Windows 编程基础。有以下几个要点:

- 学习了 Windows 基本工作原理及如何将我们自己的程序嵌入 Windows 系统中。
- 学习了一些 Windows 的基本编程概念。
- 了解了 WinMain 的重要性。
- 编写了一个简单的能够在消息框中显示文本的 Windows 程序。

## 1.5 复习测验

下面是一些复习测验题, 可帮助你跳出框框来思考并且记忆本章中所涵盖的信息。这些问题的答案可在附录 C 中找到。

- 1) Windows 2000 和 XP 使用哪种类型的多任务方式, 是抢占式还是非抢占式?
- 2) 本书主推的编译器是哪个 (虽然程序与任何 Windows 编译器都兼容)?
- 3) Windows 通知程序有事件发生, 使用什么方案?
- 4) 如果一个程序使用多个独立的部分一起工作来完成一项任务 (或者完成独立的任务), 那么这一过程叫什么?
- 5) 什么是 Direct3D?
- 6) hWnd 变量代表的是什么?
- 7) hDC 变量代表的是什么?
- 8) Windows 程序的主函数的名称是什么?
- 9) 窗口事件回调函数的名称是什么?
- 10) 用于在程序窗口中显示消息的函数是什么?

## 1.6 自己动手

这些练习将给大家带来挑战, 让你学习更多与本章给出的主题有关的知识, 帮助你提高自己的独立实践能力。

习题 1 HelloWorld 在文本框中显示一条简单的消息并显示一个感叹号图标。修改程序, 使它显示问号图标。

习题 2 现在, 修改 HelloWorld 程序, 让它在消息框中显示你的名字。



## 第2章 侦听 Windows 消息

第1章简要地讲解了 WinMain 和 WinProc 并且演示了一个简单的 Windows 程序。本章将更详细地学习 Windows 消息机制和主循环，并编写一个完整的能够在屏幕上显示一些内容的窗口程序。将学习窗口句柄和设备环境如何一起在窗口上产生输出。我们将借此巩固所掌握的基础 Windows 编程模型，也将对 Windows GDI（图形设备接口）稍作介绍，了解为什么它更适合于应用程序而不是游戏（对于游戏我们有 DirectX！）的原因。本章继续探究实时游戏循环，尤其是获得 WinMain 之外的实时循环的方法。我们将在本章学到一些新的技巧，以便让实时循环运行，为下一章的 DirectX 做准备。在学习完本章之后，我们将学到编写能够驱动本书其余代码的游戏循环的方法。所以，一定要集中注意力！

本章将学到：

- 如何创建窗口。
- 如何在窗口上绘制文本。
- 如何在窗口上绘制像素点。
- WM\_PAINT 事件在 WinProc 回调函数中如何工作。
- 如何创建实时游戏循环。
- 如何在 WinMain 中调用其他与游戏有关的函数。
- 如何使用 PeekMessage 函数。
- 如何使用 GDI 绘制位图。

### 2.1 编写一个真正的 Windows 程序

好了，让我们使用上一章所学的新知识来编写一个稍微复杂一点的程序，让它实际创建一个标准窗口并在这个窗口上绘制文本和图形，如图 2-1 所示。我们在上一章写的第一个程序是个蹩脚的程序！现在我们要写一些真正的 Windows 代码。似乎很简单，对吧？是，的确是。在窗口上绘图需要许多起始代码，我们通过示例来学习。

创建一个名为 WindowTest 的 Win32 项目并在项目中添加新的 main.cpp 文件。这里先将一个功能更完备的 Windows 程序的完整代码列出来，然后再对这个程序做反向工程并一行一行详细地讲解。在键入代码时可试着边键入边理解。如果不想键入程序，那么可从 CD-ROM 中的 \sources\chapter02\WindowTest 打开项目。

**建议 提醒：**设置 Visual C++ 的方法可参阅附录 A。创建及配置新项目的方法不在此赘述。

每章都包括一个准备好的、带有名为 main.cpp 的空白 C++ 文件的 Visual C++ 项目，以便读者在创建新项目时快速打开并使用。建议读者在学习每章内容时，为每个要创建的新项目制作一个示例项目文件夹的副本。



图 2-1 WindowTest 程序

在编译运行程序之后，应该看到如图 2-1 所示的输出。不知道如何编译程序吗？没问题，我来告诉你。最简单的方法就是按 F5 键来生成并运行程序（假设没有错误）。如果只想编译代码，那么按 Ctrl+Shift+B 组合键（生成）。这是对代码进行测试的“专业”方法——先生成，确认没有错误后再按 F5 键来运行。这些动作也可通过 Build 菜单和 Debug 菜单来执行。

```
/**
    Beginning Game Programming, Third Edition
    Chapter 2
    WindowTest program
**/

#include <windows.h>
#include <iostream>
using namespace std;

const string ProgramTitle = "Hello Windows";
/**
    ** The window event callback function
    **/
LRESULT CALLBACK WinProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    string text = "Hello Windows!";

    switch (message)
    {
        case WM_PAINT:
        {
            //get the dimensions of the window
            RECT rt;
            GetClientRect(hWnd, &rt);

            //start drawing on device context
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hWnd, &ps);
```

```

        //draw some text
        DrawText(hdc, text.c_str(), text.length(), &rt, DT_CENTER);

        //draw 1000 random pixels
        for (int n=0; n<3000; n++)
        {
            int x = rand() % (rt.right - rt.left);
            int y = rand() % (rt.bottom - rt.top);
            COLORREF c = RGB(rand()%256, rand()%256, rand()%256);
            SetPixel(hdc, x, y, c);
        }

        //stop drawing
        EndPaint(hWnd, &ps);
    }
    break;

    case WM_DESTROY:
        PostQuitMessage(0);
        break;
}
return DefWindowProc(hWnd, message, wParam, lParam);
}

/**
** Helper function to set up the window properties
**/
ATOM MyRegisterClass(HINSTANCE hInstance)
{
    //set the new window's properties
    WNDCLASSEX wc;
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC)WinProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = NULL;
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = ProgramTitle.c_str();
    wc.hIconSm = NULL;
    return RegisterClassEx(&wc);
}

/**
** Helper function to create the window and refresh it
**/
bool InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    //create a new window
    HWND hWnd = CreateWindow(
        ProgramTitle.c_str(),           //window class

```

```

    ProgramTitle.c_str(),           //title bar
    WS_OVERLAPPEDWINDOW,           //window style
    CW_USEDEFAULT, CW_USEDEFAULT,   //position of window
    640, 480,                       //dimensions of the window
    NULL,                           //parent window (not used)
    NULL,                           //menu (not used)
    hInstance,                      //application instance
    NULL);                          //window parameters (not used)

//was there an error creating the window?
if (hWnd == 0) return 0;
    //display the window
    ShowWindow(hWnd, nCmdShow);
    UpdateWindow(hWnd);

    return 1;
}

/**
** Entry point for a Windows program
**/
int WINAPI WinMain(HINSTANCE hInstance,
                  HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine,
                  int nCmdShow)
{
    //create the window
    MyRegisterClass(hInstance);
    if (!InitInstance (hInstance, nCmdShow)) return 0;

    // main message loop
    MSG msg;
    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}

```

好了，这是 WindowText 程序的完整清单，是我们的第一个完整的 Windows 程序，它能够显示一个标准的程序窗口。现在，我们对其做反向工程，看看 Windows 程序是如何工作的。

### 2.1.1 理解 InitInstance

InitInstance 是 WinMain 调用的第一个函数，用于创立程序。InitInstance 基本上只创建程序窗口。这个函数的代码本可直接插入到 WinMain 中，但将它们放到单独的函数中会更方便（这与多实例的处理有关，因为一个程序可能会多次运行）。注意，InitInstance 不是一个像 WinMain 那样的基本 Windows 函数，而只是一个“助手”函数。实例句柄是一个程序中使用的全局变量，用

于保存主实例。下面展示典型的 InitInstance 函数调用的形式及它做的工作。不过无须将其当成律条来遵守，因为这只是个标准做法而已，并不是要求。

### 1. InitInstance 函数调用

InitInstance 的函数调用如下所示：

```
bool InitInstance( HINSTANCE hInstance,
                  int         nCmdShow )
```

我们来了解一下参数：

- HINSTANCE hInstance。WinMain 传递的第一个参数，是它从 Windows 接收来的程序实例。InitInstance 将使用全局实例来检查这个参数，看看新实例是否需要终止（Windows 中的常见过程）。如果是，那么程序的主实例会被设置为前台窗口。对于用户而言，就好像再次运行程序的结果就是将原来的实例提到前面来。
- int nCmdShow。WinMain 传递给 InitInstance 的第二个参数，也是从 Windows 接收的参数。这个参数最常见的值有：SW\_HIDE 和 SW\_SHOW，Windows 通常依据操作系统中的事件（例如电源关闭）来发送这个值。

InitInstance 函数返回一个布尔值，它要么是 1（true），要么是 0（false），告诉 WinMain 启动是成功了还是失败了。注意，WinMain 没有将任何命令行参数传递给 InitInstance。如果想处理 lpCmdLine 字符串，则可创建一个新函数来处理它，也可按照通常的做法来做，即在 WinMain 中处理参数。

### 2. InitInstance 的结构

在应用程序编程中，经常推荐使用资源表来处理字符串。资源字符串的使用实际上是个人偏好问题（而笔者不使用它们）。有时需要将游戏中的文本移植到另一种语言，而这正是将字符串储存为资源所能带来的便利。但总的来说，这种用法并不普遍。显示资源中的简单消息的代码需要查找每个用到的字符串，这会降低程序运行速度并让代码更为凌乱，尤其是对初学者而言。

从代码看，InitInstance 函数颇为简单。下面先列出代码，然后讲解函数的每一部分：

```
bool InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    //create a new window
    HWND hWnd = CreateWindow(
        ProgramTitle.c_str(),           //window class
        ProgramTitle.c_str(),           //title bar
        WS_OVERLAPPEDWINDOW,           //window style
        CW_USEDEFAULT, CW_USEDEFAULT,  //position of window
        640, 480,                       //dimensions of the window
        NULL,                           //parent window (not used)
        NULL,                           //menu (not used)
        hInstance,                      //application instance
        NULL);                          //window parameters (not used)

    //was there an error creating the window?
    if (hWnd == 0) return 0;
```



```

//display the window
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);

return 1;
}

```

注意，在这段代码之前，程序实际上根本没有用户界面！使用 `CreateWindow` 函数创建的主窗口成为程序所用的窗口。`InitInstance` 的全部工作就是创建应用程序所需的新窗口并显示。`CreateWindow` 的参数列表中包括了描述每个参数用途的注释。在创建（并校验）了窗口之后，最后几行代码实际显示新创建的窗口：

```

ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);

```

`hWnd` 值由 `CreateWindow` 函数传递给这些函数。在创建窗口时，窗口就已经存在于 Windows 中了，只是看不见。`UpdateWindow` 通过将 `WM_PAINT` 消息发送给窗口处理器告诉新窗口把自己绘制出来。不仅如此，程序也经常以这种方式和自己对话，这在 Windows 编程中很常见。`InitInstance` 中的最后一行将值 1 (true) 返回给 `WinMain`：

```
return 1;
```

如果读者还记得的话，则知道 `WinMain` 对待这个返回值很严肃！如果 `InitInstance` 目前的情势不对，`WinMain` 将终止程序：

```
if (!InitInstance (hInstance, nCmdShow)) return 0;
```

从 `WinMain` 中返回一个值，无论它是 1 (true) 还是 0 (false) 都将立即终止程序。如果 `InitInstance` 的返回值是 1，那么前面讲过，`WinMain` 将继续执行，它在 `while` 循环中进行消息处理，然后程序将开始运行。

### 2.1.2 理解 `MyRegisterClass`

`MyRegisterClass` 是一个非常简单的函数，它用于设置程序所需的主窗口类的值。`MyRegisterClass` 中的代码本可放置在 `WinMain` 中。实际上，所有这些东西本来都可以塞到 `WinMain` 里面，Windows 是不会抱怨的。不过，把 Windows 程序的初始化代码分开放置到可识别的（也是标准的）助手函数中可以使程序更易理解一些，至少在学习阶段是如此。`WinMain` 调用 `InitInstance` 并且通过调用 `MyRegisterClass` 来设置程序窗口。这又是一个并非必需的可选助手函数（既然必须将代码插到某个地方，何不直接用这个函数呢？）。也可以重命名这个函数，Windows 不会介意。

#### 1. `MyRegisterClass` 函数调用

`InitInstance` 将两个参数传递给 `MyRegisterClass`，以便设置窗口类：

```
ATOM MyRegisterClass( HINSTANCE hInstance )
```

我们对这些参数都已熟悉了。`hInstance` 就是那个由 `WinMain` 传递给 `InitInstance` 的实例。这

个变量到处可见！读者还记得吧，hInstance 储存运行中的程序的当前实例，InitInstance 会将它的值复制到一个全局变量中。第二个参数很容易理解，它是 InitInstance 中以 char \* 类型创建的值，且被初始化为窗口类名称（本例中为“Hello World”）。注意，这也可以是个 Unicode 字符串。

**建议** MyRegisterClass 返回的 ATOM 数据类型被定义为 WORD，而 WORD 在 Windows 的一个头文件中被定义为 unsigned short。为什么不直接使用 unsigned short 呢？笔者还真不知道。如果你想这么做当然可以！笔者敢打赌 Microsoft 里甚至都没人记得这个 ATOM 是怎么来的。

## 2. MyRegisterClass 的要点

这个函数的要点是什么？下面再次列出 MyRegisterClass() 以便参考。在代码清单之后将详细讲解这个函数。这些属性的使用对于 Windows 程序来说简直就像家务活。笔者对此不太关心的原因是当 DirectX 登场之后我们将占有窗口。所以，有谁会去关心这个注定很快会被硬件 polygon 所占领的窗口要用哪个特殊属性呢？不过，在初期讲解所有这些基础知识很重要。笔者从大师 Charles Petzold 那里学习了《Windows 编程》<sup>①</sup>，而现在，通过间接的方式，该轮到你向他学习了！

```
ATOM MyRegisterClass(HINSTANCE hInstance)
{
    //set the new window's properties
    WNDCLASSEX wc;
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC)WinProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = NULL;
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = ProgramTitle.c_str();
    wc.hIconSm = NULL;
    return RegisterClassEx(&wc);
}
```

首先，MyRegisterClass 定义了一个新变量 wc，类型是 WNDCLASS。这个结构的每个成员都按顺序在 MyRegisterClass 中定义了，所以没必要列出这个结构。

窗口样式 wc.style 被设置成 CS\_HREDRAW | CS\_VREDRAW。管道符号 (|) 代表一种位合并的方法。CS\_HREDRAW 值使得程序窗口在移动或尺寸调整改变宽度时完全重新绘制。同样的，CS\_VREDRAW 使得窗口高度调整后完全重新绘制。

wc.lpfnWinProc 变量需要多做些解释，因为它不是一个简单的变量，而是一个指向一个回调函数的长指针。这是非常重要的，要是不设定这个值，消息就无法传递给程序窗口 (hWnd)。

<sup>①</sup> Charles Petzold 是著名的《Windows 编程》一书的作者。这是一本 Windows 编程的经典入门书籍。——译者注



当 Windows 消息和这个 `hWnd` 的值匹配时, 回调窗口过程会自动被调用。所有的消息都是如此, 包括用户输入和窗口重绘。任何按钮的按下、屏幕的刷新或其他事件的发生都会经历这一回调过程。这个函数可以起任意名称, 例如 `BigBadGameWindowProc`, 只要它的返回值是 `LRESULT CALLBACK` 而且参数正确即可。

结构变量 `wc.cbClsExtra` 和 `wc.cbWndExtra` 在大多数时候应设为零。这些变量只用来为窗口过程多增加一些额外的内存空间, 我们实在是无需使用它们。

`wc.hInstance` 设为传递给 `MyRegisterClass` 的 `hInstance` 参数值。主窗口需要知道正在使用的是哪个实例。如果真想让程序晕菜, 不妨将每个新实例设置为指向同一个程序窗口。这会很有趣! 不过这应该不可能发生, 因为游戏的新实例会被终止, 不允许运行。

`wc.hIcon` 和 `wc.hCursor` 可以很好地自解释。`LoadIcon` 函数通常用于从资源中装载一个图标图像, 而 `MAKEINTRESOURCE` 宏为资源标识符返回一个字符串值。这个宏在游戏中不常使用 (除非游戏需要在窗口中运行)。

`wc.hbrBackground` 被设置为用于绘制程序窗口背景的一个刷子句柄。默认使用的是后援对象 `WHITE_BRUSH`。它可以是一个位图图像、一个自定义刷子或者任何一种颜色。

`wc.lpszMenuName` 被设置为程序菜单的名称, 这也是一个资源。本书中的示例程序不使用菜单。

`wc.lpszClassName` 被设置为传递给 `MyRegisterClass` 的 `szWindowClass` 参数值。它给窗口指定特定的类名称, 和 `hWnd` 一起用在消息处理中。

最后, `MyRegisterClass` 调用 `RegisterClassEx` 函数。设置了窗口细节的 `WNDCLASS` 变量 `wc` 传递给了这个函数。如果返回值为零表示失败; 如果成功地将窗口注册到了 Windows, 那么这个值将传回给 `InitInstance`。

唉, 这些是否搅得你头疼? 这里不指望读者现在就能够记住所有这些消息, 不过作为游戏程序员, 理解所有的工作原理总是不错的, 这样能最大地发挥出所使用的硬件的能力。

**建议** `InitInstance` 和 `MyRegisterClass` 函数中的代码并不是一定要位于不同的函数中。可以将这些代码直接放在 `WinMain` 中, 我们在后面的章节中就是这么做的。但目前, 分成多个小步骤来实现有助于理解 Windows 编程。

### 2.1.3 晒一晒 WinProc 的秘密

`WinProc` 是窗口回调过程, Windows 通过它将事件传递给程序。回调函数是被调用回来的函数 (可能你自己已经琢磨出了这句话的意思)。回忆一下 `MyRegisterClass` 设置的传递给 `RegisterClassEx` 的 `WNDCLASS` 结构。一旦注册了类, 就可创建窗口并显示在屏幕上。该结构中的一个字段, `lpfnWinProc`, 被设置为窗口回调过程的名称, 通常为 `WinProc`。这个函数处理所有发送给主程序窗口的消息。所以, 通常 `WinProc` 是主程序源代码文件中最长的函数。图 2-2 显示了 `WinProc` 处理事件消息的方法。

#### 1. WinProc 函数调用

窗口回调函数如下所示:

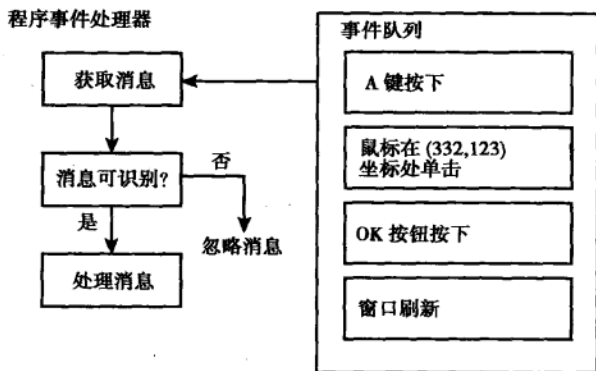


图 2-2 WinProc 回调函数处理与该应用程序相关的事件

```

LRESULT CALLBACK WinProc( HWND hWnd,
                          UINT message,
                          WPARAM wParam,
                          LPARAM lParam )
  
```

需要了解这个函数，因为它是通向初始化 Direct3D 的钥匙。它的参数既简单又直白，它们代表 Windows 程序的真实“引擎”。据前述，这一信息是早先在 WinMain 中由 GetMessage 函数获取的。不要将 InitInstance 和 WinProc 混淆在一起。InitInstance 只运行一次，对选项进行设置。而后就由 WinProc 接管，接收并处理消息。

我们来看看 WinProc 的参数：

- **HWND hWnd**。第一个参数是窗口句柄。在游戏中，通常要使用 hWnd 作为参数创建一个新的设备环境句柄，也就是一个 hDC。在 DirectX 到来之前，必须要保留好窗口句柄，因为只要引用一个窗口或控件就必须使用到它。在 DirectX 程序中，窗口句柄仅在开始时用于创建窗口。
- **UINT message**。第二个参数是发送给窗口回调过程的消息。消息可以是任何东西，甚至是无需使用的消息。由于这个原因，有一个将消息传递给默认消息处理器的方法（在下一节讨论）。
- **WPARAM wParam** 和 **LPARAM lParam**。最后两个参数是与特定命令消息一起传递过来的参数值的高位和低位。下一节将讲解它们。

## 2. WinProc 的大秘密

在随后的章节中开发的助手函数，其目标之一是为 WinProc 中的初始化和消息处理等事务提供帮助。我们的游戏库中的函数将在单独的源代码文件中处理窗口消息，以便将 Windows 核心代码与游戏代码分开（这样更易使用）。

以下是窗口回调过程的一个简单版本，随后是其解释：

```

LRESULT CALLBACK WinProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    string text = "Hello Windows!";
  
```

```

switch (message)
{
    case WM_PAINT:
    {
        //get the dimensions of the window
        RECT rt;
        GetClientRect(hWnd, &rt);

        //start drawing on device context
        PAINTSTRUCT ps;
        HDC hdc = BeginPaint(hWnd, &ps);

        //draw some text
        DrawText(hdc, text.c_str(), text.length(), &rt, DT_CENTER);

        //draw 1000 random pixels
        for (int n=0; n<3000; n++)
        {
            int x = rand() % (rt.right - rt.left);
            int y = rand() % (rt.bottom - rt.top);
            COLORREF c = RGB(rand()%256, rand()%256, rand()%256);
            SetPixel(hdc, x, y, c);
        }

        //stop drawing
        EndPaint(hWnd, &ps);
    }
    break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
}
return DefWindowProc(hWnd, message, wParam, lParam);
}

```

由于我们已经熟悉了这些参数，就直接切入正题了。这个函数可以分为两个主要部分：声明部分和如同一个大的 if 语句嵌套的 switch 语句。在 switch 语句中也有两个主要部分：处理命令消息和正常消息的 case 语句。命令将使用 WinProc 的最后两个参数：wParam 和 lParam，而正常消息通常不需要这些参数。

PAINTSTRUCT 变量 ps 用在 WM\_PAINT 消息处理器中，用于启动及停止屏幕更新，这就如在进行更新时解锁和锁住设备环境（于是在这个过程中屏幕内容不会不完整）。hdc 变量也在 WM\_PAINT 消息处理器中使用，用于获取程序窗口的设备环境。其他变量用于在屏幕上显示消息 (szHello) 及在窗口中绘制像素点 (x、y、n 和 c)。

在变量声明之后是 switch(message) 语句。它是处理多个消息更为容易的方法，要比使用嵌套 if 语句好得多。在处理大量条件测试时，switch 的能力要强得多，所以将它用在 WinProc 中进行消息的检查。

首先要讲解的是 WM\_DESTROY。WM\_DESTROY 消息标识符告诉窗口该关闭了。程序应该从内存中移除对象，然后调用 PostQuitMessage 函数结束程序，从而优雅地关闭。下一步在编

写 Direct3D 代码时，这将会是唯一需要关心的消息，因为 WM\_PAINT 在 Direct3D 程序中不需要。

好了，回到第一个消息标识符——WM\_PAINT。对于游戏编程而言这绝对是个最为有趣的消息，因为窗口更新在这里处理。再次看一下 WM\_PAINT 的代码：

```
//get the dimensions of the window
GetClientRect(hWnd, &rt);

//start drawing on device context
hdc = BeginPaint(hWnd, &ps);
//draw some text
DrawText(hdc, text.c_str(), text.length(), &rt, DT_CENTER);

//draw 1000 random pixels
for (n=0; n<3000; n++)
{
    x = rand() % (rt.right - rt.left);
    y = rand() % (rt.bottom - rt.top);
    c = RGB(rand()%256, rand()%256, rand()%256);
    SetPixel(hdc, x, y, c);
}

//stop drawing
EndPaint(hWnd, &ps);
break;
```

调用 BeginPaint 函数是为了锁住设备环境以便进行更新（使用窗口句柄和 PAINTSTRUCT 变量）。对 GetClientRect 的调用是为了获取程序窗口的矩形区域并保存到一个 RECT 变量中。DrawText 使用它将消息绘制在窗口的中央。注意，BeginPaint 返回程序窗口的设备环境。在每次刷新时这都是必需的，因为，尽管不常见，但在程序运行中不能保证设备环境是个常量（例如，内存不够用，程序被转移到虚拟内存的分页文件中，而后再重新取出。这样的事件几乎肯定会生成新的设备环境）。

真正在用户界面上做了一些事情的代码是第三行，它调用 DrawText。这个函数在目标设备环境中显示消息。末尾的 DT\_CENTER 参数告诉 DrawText 将消息对齐到所传递的矩形的顶部中央位置。当然，也正是这部分代码在屏幕上绘制像素点。那么，如果更改窗口尺寸所有的像素点都会被重新绘制吗？试一试吧！很酷，对不？这完美地演示了 WM\_PAINT：在窗口需要重绘时它会被调用。如果更改窗口尺寸，会发生对 WM\_PAINT 的多次调用，每次的矩形（通过 GetClientRect 返回）都不一样。

绘制消息处理器的最后一行调用 EndPaint 以关闭本次消息处理器迭代所用的图形系统。

**建议** 系统不会连续不断地调用 WM\_PAINT，只有当窗口必须重绘时才会调用，这与实时循环不同。所以，WM\_PAINT 不是个适合为游戏插入屏幕刷新代码的位置。就如第 3 章要学的内容那样，我们必须修改 WinMain 中的循环，让代码在实时循环中运行。

## 2.2 什么是游戏循环

Windows 编程比我们在这几页中所讲解的要博大精深得多，我们所需关注的只是能够让 DirectX 启动起来的有限代码。一个“真正”的 Windows 应用程序应该有一个菜单、一个状态栏、一个工具栏及许多对话框，这也是一般的 Windows 编程书籍又大又厚的原因。我们这里要关注的是游戏的创建而不是将大量篇幅花费在操作系统的逻辑上。我们真正想做的是逃离 Windows 代码，用上简单的、寻常的、C++ 程序里标准的 main 函数（但在 Windows 程序中没有它，而是 WinMain）。

**建议** Windows 编程方面的书籍推荐参见附录 B。

有一种方法是将所有的基础 Windows 代码（包括 WinMain）都放到一个源代码文件中（例如 winmain.cpp），然后使用另一个源代码文件（例如 game.cpp）来放游戏。然后，就是在 WinMain 中调用某种形式的 main 函数的简单事情了，“游戏代码”会在程序窗口创建之后立即开始运行。这实际上是许多系统和库的标准做法，它将操作系统进行抽象，给程序员提供一个标准的接口。Allegro 就是一种这样的库，它是跨平台的游戏库，可在 Windows、Linux、Mac 和 Unix 系统上使用。如果编写一个使用 Allegro 的程序，那么这个程序无需修改就可以在这些系统中的任何系统上编译！这就将“移植”工作带入可管理的层次，因为对于一个为 Windows 开发的程序而言，仅需一点点的努力即可为 Linux 重新编译（反之亦然）。这里不是真要往这个方向走，但这对于尽可能抽象代码如何有益是个很好的示例。

**建议** 如果想学习更多与 Allegro 游戏库相关的知识，参见附录 B 中关于这一主题的一本尖端书籍的信息！

### 2.2.1 老的 WinMain

以下是到目前为止我们一直在用的 WinMain 版本：

```
int WINAPI WinMain(HINSTANCE hInstance,
                  HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine,
                  int nCmdShow)
{
    MSG msg;
    MyRegisterClass(hInstance);
    if (!InitInstance (hInstance, nCmdShow))
        return FALSE;

    while (GetMessage(&msg, NULL, 0, 0))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    return msg.wParam;
}
```



这个版本的 WinMain 仅有一个问题：它没有一个连续的循环，只有一个有限的循环用于处理任何尚未处理的消息，然后退出（参见黑体的 while 一行）。

### 1. 对持续性的需要

当精灵或 3D 模型在屏幕上动画显示，敌方角色到处移动，枪弹和爆炸再加上高热原子核反应的爆炸在背景肆虐时，我们需要能够将 Windows 消息放在一边而一直工作的东西！简单地说，上面列出的是一个蠢笨的、无生命力的 WinMain 版本，完全不适合于游戏。我们需要这样一种东西，它能够持续运行而无论是否有事件消息进来。创建一个无论 Windows 在做什么都能一直运行的实时循环的关键就在于修改 WinMain 中的 while 循环。

首先，while 循环受一条消息的制约，而游戏必须在循环中保持运行，无论是否有消息。所以这绝对需要有所改变！图 2-3 显示了当前的 WinMain。

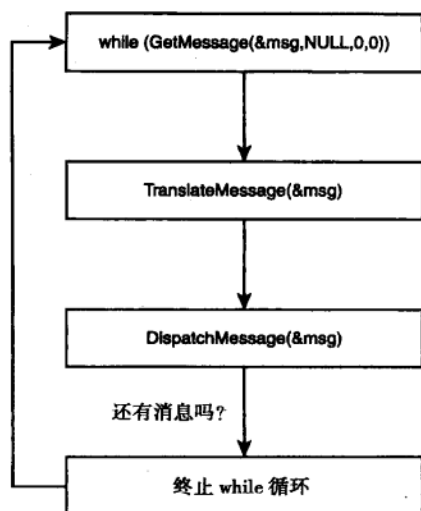


图 2-3 标准的 WinMain 对实时游戏循环不友好

### 2. 实时的终止器

注意主循环在没有消息时的终止和在尚有消息存在时持续处理的方式。如果在这个版本的 WinMain 中调用主游戏循环会发生什么情况呢？嗯，游戏循环偶尔会执行，内容会在屏幕上更新。但更经常的是什么都不会发生。这是什么原因呢？因为这是个事件驱动的 while 循环，而我们需要一个普通的、寻常的、过程的 while 循环来保持系统运行、运行、再运行，无论发生什么。实时游戏循环在游戏结束之前必须保持运行不停。为了避免大家疑惑，下一章将展示设立一致的、常规的帧速率的方法。目前的目标是让一切尽可能快地运行，而后再考虑计时的问题。我们必须先让程序跑起来，然后再进行优化或清理（如果有时间）。

现在来看另外一个图示，见图 2-4。这是新版本的 WinMain，现在它支持实时游戏循环了。它不再只是对事件进行循环。它而是不管事件如何，一直保持循环。

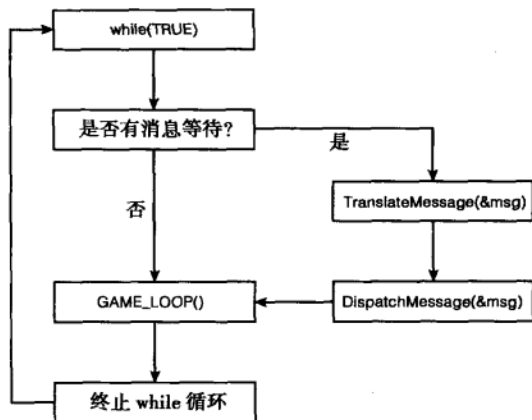


图 2-4 新修改的 WinMain 对实时游戏循环要友好得多

## 2.2.2 WinMain 和循环

制作一个实时循环的关键在于对 WinMain 中的 while 循环进行修改，让它能无限运行，然后在 while 循环内部检查消息。无限的意思是循环将永远保持运行，除非受到中断并导致循环退出（通过在循环中调用 exit 或 return）。为了使用一个无休止的循环，有一个可替代 GetMessage 函数来检测是否有事件消息进来的函数。这个函数就是 PeekMessage。顾名思义，这个函数可查看到来的消息而无需将其从消息队列中取出。

现在，我们当然不能让消息队列堆满（它最终会造成程序崩溃），无论是否有消息我们都要使用 PeekMessage 代替 GetMessage。如果有消息，没问题，继续并处理它们。否则，就将控制权交给下一行代码。不难发现，GetMessage 不是很友好的，它不让我们的游戏循环运行，除非在消息队列中有等待处理的消息存在。另一方面，PeekMessage 是友好的，在没有消息等待时它只会将控制传递给下一条语句。

### 1. PeekMessage 函数真面目

让我们来看看 PeekMessage 函数的格式：

```

BOOL PeekMessage(
    LPMSG lpMsg,          //pointer to message struct
    HWND hWnd,           //window handle
    UINT wMsgFilterMin,   //first message
    UINT wMsgFilterMax,   //last message
    UINT wRemoveMsg);    //removal flag
  
```

它的参数如下所示：

- LPMSG lpMsg。这个参数是一个指向描述本消息的消息结构（类型、参数等）的长指针。
- HWND hWnd。这是与事件关联的窗口的句柄。
- UINT wMsgFilterMin。这是已收到的第一条消息。
- UINT wMsgFilterMax。这是已收到的最后一条消息。



- `UINT wRemoveMsg`。这是用于确定在读了消息之后如何对消息进行处理的标志。它可以是 `PM_NOREMOVE`，将消息留在消息队列中；也可以是 `PM_REMOVE`，在读取消息之后将它从消息队列中移除。

## 2. 将 `PeekMessage` 插到 `WinMain` 中

好了，我们现在就用上 `PeekMessage`，这样就可以了解这一切与游戏的编写是如此之般配了。以下是 `WinMain` 中新版本的主循环，它使用了 `PeekMessage`（还有几行额外的代码，我很快就会讲解到）。

```
bool gameover = false;
while (!gameover)
{
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
    {
        //handle any event messages
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    //process game loop
    Game_Run();
}
```

可以注意到，在这个新版本的 `while` 循环中使用了对 `PeekMessage` 的调用而不是 `GetMessage`，此外还会看到 `PM_REMOVE` 参数，它用于将事件消息从队列中取出并处理。在实际情况中，没有消息会真的进入 DirectX 程序（除了 `WM_QUIT` 有可能外），因为大多数处理都发生在 DirectX 库中。

看一下寻找 `WM_QUIT` 消息的 `if` 语句。这是导致 `while` 循环退出的唯一情况；否则它就会不停地运行。

好，假设我们有一个游戏循环。那么它能做什么呢？你可能也看到了加入的那行额外代码，因为它名叫 `Game_Run`。这个函数不是 Windows 的一部分，实际上它尚未存在。很快我们就会亲自编写这个函数！在第 3 章，在我们最终有机会开始进入 DirectX 代码时，这个函数将更显示出其意义。

那就是说，我们得看一看完成的 `WinMain` 版本：

```
int WINAPI WinMain(HINSTANCE hInstance,
                  HINSTANCE hPrevInstance,
                  LPSTR lpCmdLine,
                  int nCmdShow)
{
    MyRegisterClass(hInstance);
    if (!InitInstance(hInstance, nCmdShow)) return 0;

    //initialize the game
    if (!Game_Init()) return 0;

    // main message loop
    bool gameover = false;
    while (!gameover)
    {
```

```

MSG msg;
if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
{
    //decode and pass messages on to WndProc
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}

//process game loop
Game_Run();
}
//do cleanup
Game_End();

//end program
return msg.wParam;
}

```

好的，我承认，我是超前了一些，在没有告知的情况下就偷偷添了一些新东西。这里所说的是名为 `Game_Init`、`Game_Run` 和 `Game_End` 的这几个新的未知函数。别沮丧，不拘小节地将新东西扔给你却不告知也不解释不会形成习惯。不过有时候我觉得，事先给出某些东西的工作原理，然后再讲解，会很有趣。在本例中，我是先提前一点儿进行计划。

### 状态驱动的游戏

这实际上是顽固的游戏程序员中恼人的争论来源之一。有些人认为游戏应该从一开始就是状态驱动的，所有函数调用都必须极度抽象，这样代码就能移植到其他平台。例如，有些人编写的代码会隐藏所有的 Windows 代码，于是需要编写相似的 Mac 或 Linux 版本时就有可能无需太多困难就能将大部分游戏代码移植到这些平台上。

我们将稍微探究一下这个主题，因为这是良好的开发习惯！即使在压力下要求马上完成一款游戏，即使需要一次输入 16 小时代码，只要你是个真正的专业人士，那么你也会想方设法将一些神经元留给更高层次的东西——例如编写干净的代码。我觉得最干净的代码是能够在超过一款编译器上编译的代码。

例如，本书中所有的代码——我指的是所有，都可使用 Bloodshed Software 的免费编译器 Dev-C++ 和 DirectX Devpack 一起编译。使用 Dev-C++ 所使用的 GCC 编译器可帮助我们编写更好的代码，因为这样的代码必须遵循 C++ 标准。

## 2.3 GameLoop 项目

为了展示我们所讨论的实时编程的实际应用，本节将带领大家创建一个新的包含新版本的 WinMain 和所有我们在代码清单中添加的那些新函数的项目。

按通常的方法创建一个新的 Win32 项目（如果需要帮助参见附录 A）。将新项目命名为 GameLoop，如图 2-5 所示。接下来，打开 File 菜单，选择 New 打开 New 对话框。从可用的文件列表中选择加入到项目中的文件类型为 C++ Source File（记得忽略 C++ 部分并将文件以 .c 扩展

名命名)。将新文件命名为 winmain.cpp, 然后单击 OK 按钮将文件添加到新项目中, 如图 2-6 所示。作为另一种方法, 也可从 CD-ROM 中装载 GameLoop 项目。

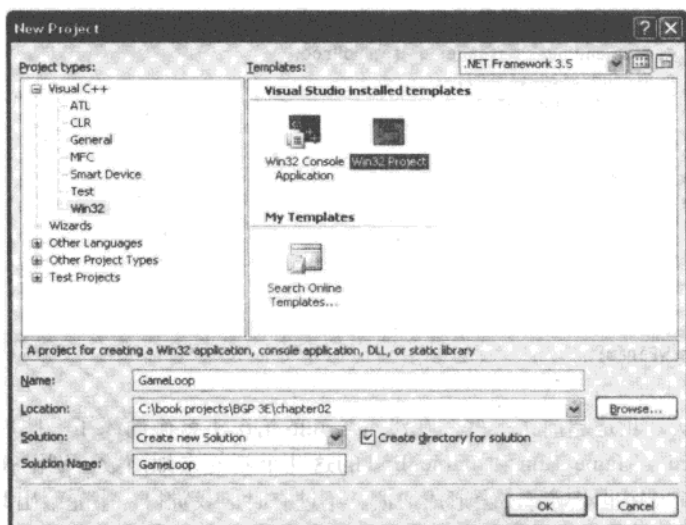


图 2-5 创建名为 GameLoop 的新 Win32 项目

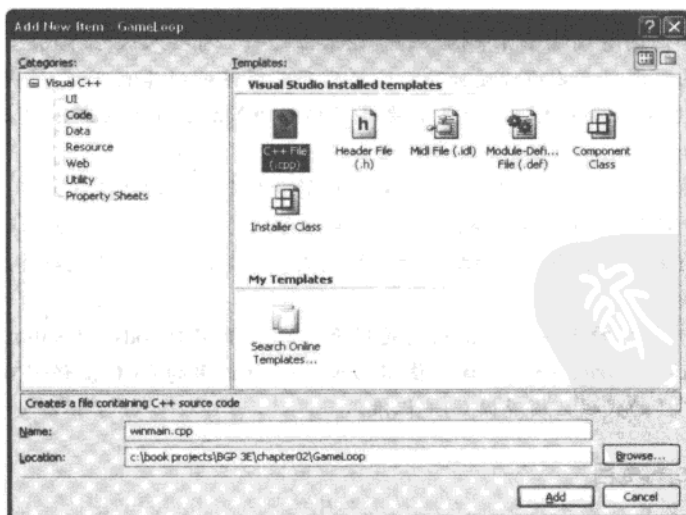


图 2-6 在项目中添加 winmain.cpp 文件

## GameLoop 程序的源代码

在这里提供的代码将会是所有后续程序的基础, 后续程序只会有很少的一些更改。在最后一

章中你会发现有好几处都是对相似代码的小改进。接着，打开 GameLoop 项目中的 winmain.cpp 文件并键入下列清单中的代码。下面会讲解它。

```
/**
    Beginning Game Programming, Third Edition
    Chapter 2
    GameLoop project
**/

#include <windows.h>
#include <iostream>
#include <time.h>
using namespace std;

const string APPTITLE = "Game Loop";
HWND window;
HDC device;
bool gameover = false;

/**
    ** Loads and draws a bitmap from a file and then frees the memory
    **/
void DrawBitmap(char *filename, int x, int y)
{
    //load the bitmap image
    HBITMAP image = (HBITMAP)LoadImage(0, "c.bmp", IMAGE_BITMAP, 0, 0,
    LR_LOADFROMFILE);

    //read the bitmap's properties
    BITMAP bm;
    GetObject(image, sizeof(BITMAP), &bm);

    //create a device context for the bitmap
    HDC hdcImage = CreateCompatibleDC(device);
    SelectObject(hdcImage, image);

    //draw the bitmap to the window (bit block transfer)
    BitBlt(
        device,                //destination device context
        x, y,                  //x,y location on destination
        bm.bmWidth, bm.bmHeight, //width,height of source bitmap
        hdcImage,              //source bitmap device context
        0, 0,                  //start x,y on source bitmap
        SRCCOPY);              //blit method

    //delete the device context and bitmap
    DeleteDC(hdcImage);
    DeleteObject((HBITMAP)image);
}

/**
    ** Startup and loading code goes here
    **/
```

```
bool Game_Init()
{
    //start up the random number generator
    srand(time(NULL));

    return 1;
}

/**
** Update function called from inside game loop
**/
void Game_Run()
{
    if (gameover == true) return;
    //get the drawing surface
    RECT rect;
    GetClientRect(window, &rect);

    //draw bitmap at random location
    int x = rand() % (rect.right - rect.left);
    int y = rand() % (rect.bottom - rect.top);
    DrawBitmap("c.bmp", x, y);
}

/**
** Shutdown code
**/
void Game_End()
{
    //free the device
    ReleaseDC(window, device);
}

/**
** Window callback function
**/
LRESULT CALLBACK WinProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_DESTROY:
            gameover = true;
            PostQuitMessage(0);
            break;
    }
    return DefWindowProc(hWnd, message, wParam, lParam);
}

/**
** MyRegiserClass function sets program window properties
**/
ATOM MyRegisterClass(HINSTANCE hInstance)
{
    //create the window class structure
```

```

WNDCLASSEX wc;
wc.cbSize = sizeof(WNDCLASSEX);
//fill the struct with info
wc.style = CS_HREDRAW | CS_VREDRAW;
wc.lpfnWndProc = (WNDPROC)WinProc;
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hInstance = hInstance;
wc.hIcon = NULL;
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH)GetStockObject(BLACK_BRUSH);
wc.lpszMenuName = NULL;
wc.lpszClassName = APPTITLE.c_str();
wc.hIconSm = NULL;

//set up the window with the class info
return RegisterClassEx(&wc);
}

```

```

/**
** Helper function to create the window and refresh it
**/

```

```

BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
    //create a new window
    window = CreateWindow(
        APPTITLE.c_str(),           //window class
        APPTITLE.c_str(),           //title bar
        WS_OVERLAPPEDWINDOW,        //window style
        CW_USEDEFAULT,              //x position of window
        CW_USEDEFAULT,              //y position of window
        640,                        //width of the window
        480,                        //height of the window
        NULL,                       //parent window
        NULL,                       //menu
        hInstance,                  //application instance
        NULL);                      //window parameters

    //was there an error creating the window?
    if (window == 0) return 0;

    //display the window
    ShowWindow(window, nCmdShow);
    UpdateWindow(window);

    //get device context for drawing
    device = GetDC(window);

    return 1;
}

```

```

/**
** Entry point function

```



```
*/
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    MSG msg;

    //create window
    MyRegisterClass(hInstance);
    if (!InitInstance (hInstance, nCmdShow)) return 0;

    //initialize the game
    if (!Game_Init()) return 0;

    // main message loop
    while (!gameover)
    {
        //process Windows events
        if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

        //process game loop
        Game_Run();
    }

    //free game resources
    Game_End();

    return msg.wParam;
}
```

### 1. 在 Windows 中绘制位图

本示例中的 DrawBitmap 函数用于绘制位图，必须承认，这个函数很慢。这个函数适合于装载完整的背景图片或者只需将图片绘制一次的小位图，但这里却用它（或者是滥用它）来重复装载并绘制来自文件中的位图。这个函数将位图文件装入内存，让其在 Windows 中运行一次，然后在窗口的某个随机位置上绘制它（使用窗口的设备环境）。

**建议** 如果用书中的代码清单来创建本程序的话，要确认将 c.bmp 文件复制到项目文件夹中。这个文件可在 \sources\chapter02\GameLoop 文件夹中找到。还可以使用任何其他位图文件，只需更改一下代码中的文件名即可。

在真实的游戏绝不可这样做，因为这个可怜的位图文件在每次循环都需装载一次！这会慢得让人抓狂，而且又那么浪费。不过作为演示这没什么问题，因为所有与位图相关的代码都位于这个函数中，而不是散播在清单的其他地方。我希望大家专注于游戏循环和支持函数，而不是这个过了本章就用不着的古董般的位图代码。

```

void DrawBitmap(char *filename, int x, int y)
{
    //load the bitmap image
    HBITMAP image = (HBITMAP)LoadImage(0,"c.bmp",IMAGE_BITMAP,0,0,
    LR_LOADFROMFILE);

    //read the bitmap's properties
    BITMAP bm;
    GetObject(image, sizeof(BITMAP), &bm);

    //create a device context for the bitmap
    HDC hdcImage = CreateCompatibleDC(device);
    SelectObject(hdcImage, image);

    //draw the bitmap to the window (bit block transfer)
    BitBlt(
        device,                //destination device context
        x, y,                  //x,y location on destination
        bm.bmWidth, bm.bmHeight, //width,height of source bitmap
        hdcImage,              //source bitmap device context
        0, 0,                  //start x,y on source bitmap
        SRCCOPY);              //blit method

    //delete the device context and bitmap
    DeleteDC(hdcImage);
    DeleteObject((HBITMAP)image);
}

```

**建议** 如果本书是关于 Windows GDI (图形设备接口) 编程的, 那么肯定要非常详细地为你讲解所有的 GDI 图形函数! 但这里我们只是一笔带过。

为了重复绘制位图, Game\_Run 函数将位图文件名和一个随机的 (x, y) 位置 (其范围在窗口宽度和高度范围之内) 传递给 DrawBitmap 函数:

```

void Game_Run()
{
    if (gameover == true) return;

    //get the drawing surface
    RECT rect;
    GetClientRect(window, &rect);

    //draw bitmap at random location
    int x = rand() % (rect.right - rect.left);
    int y = rand() % (rect.bottom - rect.top);
    DrawBitmap("c.bmp", x, y);
}

```

## 2. 运行 GameLoop 程序

万事俱备! 现在开始运行程序。应该可以看到一个塞满了一大堆失控的疯狂的 C 位图的窗口, 如图 2-7 所示。



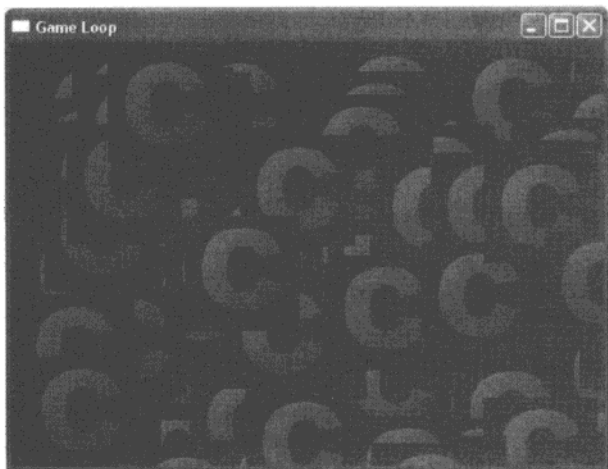


图 2-7 GameLoop 程序窗口充满了 C 位图

Windows GDI, 这个为我们提供窗口句柄和设备环境, 并且允许我们在窗口上绘图, 从而构建用户界面 (或者不使用 DirectX 的游戏) 的系统, 不客气地说, 是一种退步。我想一直向前行进, 只讲解 Windows 编程中那些为 DirectX 提供基础的方面, 忽略我单方面对此问题的评价。

## 2.4 你所学到的

本章学习了 Windows 编程基础, 以为 DirectX 编码做准备。有以下几个要点:

- 学习了更多 Windows 编程概念。
- 编写了一个使用 Windows GDI 绘制位图的简单程序。
- 学习了如何在窗口中绘制像素点。
- 剖析了一个完整的 Windows 程序并学习了它的工作原理。
- 学习了 PeekMessage 函数。
- 学习了如何修改 WinMain 中的主循环。
- 添加了一些能够让编写游戏更简单的新函数。

## 2.5 复习测验

下面是一些复习测验题, 可帮助你跳出框框来思考并且记忆本章中所涵盖的信息。这些问题的答案可在附录 C 中找到。

- 1) WinMain 函数是做什么的?
- 2) WinProc 函数是做什么的?
- 3) 程序实例是什么?
- 4) 可用于在窗口中绘制像素点的是什么函数?

- 5) 可用于在程序窗口中绘制文本的是什么函数?
- 6) 什么是实时游戏循环?
- 7) 在游戏中为什么需要使用实时循环?
- 8) 用于创建实时循环的主要的助手函数是什么?
- 9) 哪个 Windows API 函数可用于在屏幕上绘制位图?
- 10) DC 代表的是什么?

## 2.6 自己动手

这些练习将给你带来挑战, 促使你学习更多与本章给出的主题有关的知识, 帮助你提高自己的独立实践能力。

习题 1 WindowTest 程序中的窗口背景是白色的 (WHITE\_BRUSH)。修改程序, 让它使用黑色背景。

习题 2 修改 GameLoop 程序, 让它只绘制一个能够在窗口中到处移动的位图。(提示: 需要确保位图不“飞出”窗口边界。)



## 第3章 初始化 Direct3D

本章将介绍如何编写初始化 DirectX 并且创建 Direct3D 设备的程序。Direct3D 设备用于访问视频卡的主表面（primary surface）或者帧缓冲区（frame buffer）及后台缓冲区（back buffer，用于生成平滑的图形）。后续的章节将通过对位图和表面的探究，带领你更深入地进入 Direct3D 的架构中。在开始在屏幕上装载 3D 对象并且渲染动画角色之前，我们需要了解 DirectX 的基础。

本章将学到：

- 如何初始化 Direct3D 对象。
- 如何创建用于访问视频显示的设备。
- 如何创建用于不抖动图形效果的后台缓冲区。
- 如何在窗口模式下运行 Direct3D 程序。
- 如果以全屏模式运行 Direct3D 程序。

### 3.1 初识 Direct3D

为了使用 Direct3D 或者 DirectX 的任何其他组件，必须熟悉使用头文件和库文件的方法（在 C 编程中是家常便饭），因为 DirectX 函数调用都储存在头文件中，而预编译的 DirectX 函数都储存在库文件中。例如，Direct3D 函数储存在 d3d9.lib 中，要让程序“看到”Direct3D，需要在源代码文件中使用 `#include <d3d9.h>` 指令将 d3d9.h 头文件包括进来。

这里假定你已经安装了 DirectX 9 软件开发工具包（SDK）的 Visual C++ 版或者单独的 Dev-C++ 版。如果尚未安装其中之一，在继续阅读之前需先安装。DirectX 9 SDK 位于本书配套 CD-ROM 的 \DirectX 文件夹中，Dev-C++ 位于 \dev-cpp 文件夹中。在需要设定 DirectX Runtime Support（运行时支持）选项时，需要安装调试版以便开发。

好了，可以开始了吧？

#### 3.1.1 Direct3D 接口

为了编写使用 Direct3D 的程序，必须创建一个 Direct3D 接口变量和一个图形设备变量。Direct3D 接口名为 LPDIRECT3D9，而设备对象名称为 LPDIRECT3DDEVICE9。创建变量的方法如下：

```
LPDIRECT3D9      d3d      = NULL;
LPDIRECT3DDEVICE9 d3ddev = NULL;
```

LPDIRECT3D9 对象是 Direct3D 库的大老板，这个对象控制所有的一切；而 LPDIRECT3DDEVICE9 代表的是视频卡。你或许能从这些对象的名称看出它们的来历。LP 的意思是“长指针”，于是 LPDIRECT3D9 是指向 DIRECT3D9 对象的长指针。这些定义位于 d3d9.h 头文件中，这个头文件必须包括（`#include`）在我们的源代码文件中。以下是 LPDIRECT3D9 的定义：

```
typedef struct IDirect3D9 *LPDIRECT3D9;
```

如果对指针不够熟悉，会对以上定义感到困惑。对大多数要掌握 C 的程序员指针肯定是最大的障碍。当我对某些东西不理解时，我会让我的潜意识来做功——因为我的意识有时候不是那么通畅。严肃地说，如果不了解它们，那么就直接使用这些指针和对象好了，给自己一些时间，会慢慢理解的。认为在使用某种东西之前必须要理解这种东西的工作原理，是程序员经常犯的错误之一。不用这样！昂首前进并编写 Direct3D 程序吧，你不需要立刻知道与 3D 建模或者渲染有关的任何知识。实践出真知，这会弥补对理解的缺乏。

IDirect3D9 是个接口，所以，LPDIRECT3D9 是指向 Direct3D9 接口的长指针。LPDIRECT3DDEVICE9 也是如此，它是指向 IDirect3DDevice9 接口的长指针。

### 3.1.2 创建 Direct3D 对象

下面，给出初始化主 Direct3D 对象的方法：

```
d3d = Direct3DCreate9(D3D_SDK_VERSION);
```

这个代码初始化 Direct3D，也就是说 Direct3D 准备好可使用了。首先，需要创建 Direct3D 将要输出显示内容的设备。这时 d3ddev 变量就派上用场了（注意，使用 d3d 来调用这个函数）：

```
d3d->CreateDevice(
    D3DADAPTER_DEFAULT,           //use default video card
    D3DDEVTYPE_HAL,               //use the hardware renderer
    hWnd,                         //window handle
    D3DCREATE_SOFTWARE_VERTEXPROCESSING, //do not use T&L (for compatibility)
    &d3dpp,                        //presentation parameters
    &d3ddev);                      //pointer to the new device
```

#### 硬件 T&L（变换与光照）

如果你是个技术爱好者（也就是喜欢用小机械修修补补的人），或者是个中坚的游戏者，喜欢争论视频卡的规格，那么 D3DCREATE\_SOFTWARE\_VERTEXPROCESSING 参数可能会惹怒你。如果你对视频卡一无所知，那么这就没什么问题！不过我觉得你应该是那种将追求最新计算机技术当成习惯的人，对吧？嗯，我们都知道“变换与光照”在多年前是个时髦的词，从那时开始所有的视频卡都带有 T&L——而最新的发展是可编程着色器。这一切真正的意思是，大部分的 3D 设置工作是由视频卡本身来处理的，而不是由计算机的中央处理器（CPU）来处理。

当 3Dfx 带着全世界首款 PC 用 3D 加速器卡横空出世，它给游戏界带来了风暴与革命。虽然这种事情迟早都会发生，但 3Dfx 是第一个吃螃蟹的人，因为这家公司已经为街机制造 3D 硬件多年。我记得第一次看到运行在 3D 加速状态下的 Quake 时，我张大了嘴巴。

其时，渲染管线内置到了 3D 卡中，以替代 CPU 的功能。革命持续了许多年，视频卡的多边形处理能力和功能集迅速膨胀。nVidia 随后引领了下一次革命，它将 3D 渲染管线的变换与光照处理阶段加入到了 3D 芯片本身中，将这个工作的重担从 CPU 的身上卸下。

那么，变换和光照是什么呢？变换是对多边形的操纵，而光照则如其名——给这些多边形增加光照效果。3D 芯片最初通过在硬件中渲染带纹理的多边形来增强游戏效果（这极大地改进了质量和速度），而 T&L 则通过让 3D 芯片操纵及照射场景，带来终极改进。这些都解放了 CPU，让其有时间执行其他任务，例如人工智能和游戏物理——可能你还没注意到，它在最近几年其实已经开始出现了！这并不是因为我们有更快的 CPU，而主要是由于 GPU 卸下了重担。

CreateDevice 的最后两个参数指定设备参数 (d3dpp) 及设备对象 (d3ddev)。d3dpp 必须先定义后使用，所以我们得讲讲它。可以为设备指定许多选项，如表 3-1 所示。

首先，创建一个 D3DPRESENT\_PARAMETERS 结构变量用于设置设备参数：

```
D3DPRESENT_PARAMETERS d3dpp;
```

然后在使用前将结构中的所有值清为零：

```
ZeroMemory(&d3dpp, sizeof(d3dpp));
```

表 3-1 Direct3D 呈现参数

变 量	类 型	描 述
BackBufferWidth	UINT	后台缓冲区宽度
BackBufferHeight	UINT	后台缓冲区高度
BackBufferFormat	D3DFORMAT	后台缓冲区格式，在窗口模式中，传递 D3DFMT_UNKNOWN 以使用桌面格式
BackBufferCount	UINT	后台缓冲区数量
MultiSampleType	D3DMULTISAMPLE_TYPE	全屏反走样 (anti-aliasing) 的多采样层 (multi-sampling level) 数量。通常传递 D3DMULTISAMPLE_NONE
MultiSampleQuality	DWORD	多采样的质量级别，通常传递 0
SwapEffect	D3DSWAPEFFECT	后台缓冲区的交换模式
hDeviceWindow	HWND	本设备的父窗口
Windowed	BOOL	如果是窗口模式设为 TRUE，全屏模式设为 FALSE
EnableAutoDepthStencil	BOOL	允许 D3D 控制深度缓冲区（通常设置为 TRUE）
AutoDepthStencilFormat	D3DFORMAT	深度缓冲区的格式
Flags	DWORD	选项标志（通常设为 0）
FullScreen_RefreshRateInHz	UINT	全屏刷新率（对于窗口模式必须是 0）
PresentationInterval	UINT	控制缓冲区交换速率

d3dpp 结构中有许多选项，其中还有许多子结构。本章将讲解用得到的选项，不会每个选项都讲到（信息太多了）。下面给 d3dpp 结构填充几个窗口 Direct3D 程序运行需要的值：

```
d3dpp.Windowed = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;
```

在填充了这几个值之后，就可调用 CreateDevice 创建主 Direct3D 绘图表面。

### 3.1.3 让 Direct3D 转起来

我们来创建一个关于 Direct3D 的示例项目，感受一下完整的 Direct3D 程序是如何工作的。创建一个新的 Win32 项目类型的程序，命名为 Direct3D\_Windowed（或者喜欢的任何名称，但这个名称是 CD-ROM 上的项目名）。在空白项目中添加一个新文件，命名为 winmain.cpp。现在我们来配置这个 Direct3D 项目。

**建议** 虽然文件名都使用 .cpp 作为扩展名，但这些基本上只是 C 代码（不是 C++）。Visual C++ 在某些情况下会报告不以 .cpp 结尾的源文件。

#### 1. 与 Direct3D 库链接

在项目中加入库以便程序使用的方法有两种。例如，Direct3D 是 DirectX SDK（软件开发工具包）中包含的一个库。Direct3D 库文件名为 d3d9.lib，描述它的头文件是 d3d9.h。DirectX SDK 中还有其他库和头文件。我们可将 d3d9.lib 添加到项目属性中，也可添加一条 #pragma 代码让编译器（或者，更为准确地说，是链接器）在所创建的最终可执行文件中加入这个库文件。我们将使用 #pragma 方法，不过我还会将如何在 Visual C++ 中通过设置链接器将 Direct3D 库加进来的方法告诉读者。

```
#pragma comment(lib, "d3d9.lib")
```

打开 Project 菜单并选择 Properties（菜单底部的最后一个选项）。Properties 对话框如图 3-1 所示。

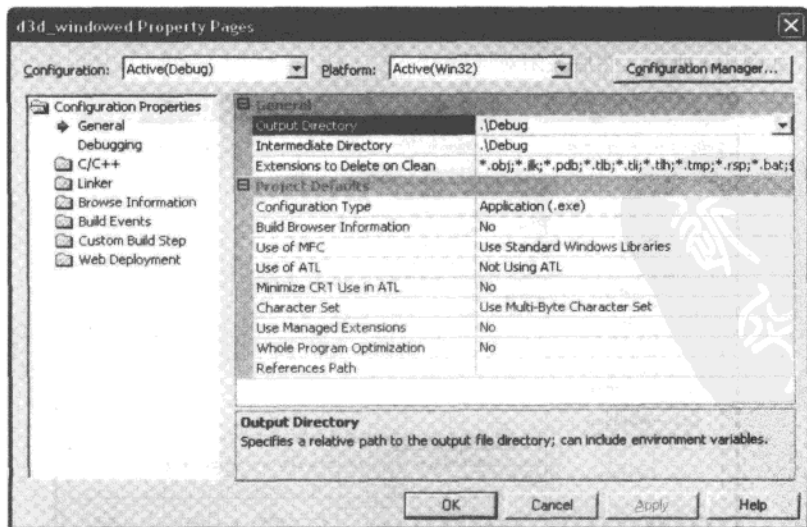


图 3-1 Microsoft Visual C++ 中 d3d\_windowed 项目的项目属性对话框

单击左边列表中的 Linker 条目，打开链接器选项。我们将注意到在 Linker 树条目下有许多子条目，例如 General、Input 和 Debugging 等。选择 Linker 下 Input 子条目，如图 3-2 所示。

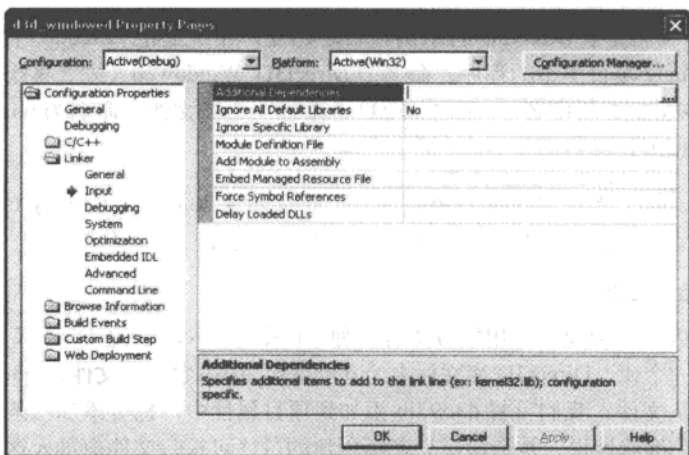


图 3-2 打开项目属性对话框中的 Link 选项卡

尤其要留意 Additional Dependencies 字段。这个字段显示了在把所有各种源代码文件编译并链接在一起来形成可执行文件时，所有链接到程序中的库文件。如果在项目中有一个 winmain.cpp 文件，那么它会被编译成 winmain.obj（这是个目标文件），它包含能够在计算机上执行的二进制指令。这是非常低级的二进制文件，是不可读的，所以不要试图打开它（在 Debug 文件夹中可见到各种不同的输出文件，在编译程序时，这个文件夹创建于程序主文件夹之中）。

现在，我们将 Direct3D 库文件添加到这个库列表中。在 Additional Dependencies 字段中加入“d3d9.lib”，如图 3-3 所示，然后关闭对话框。

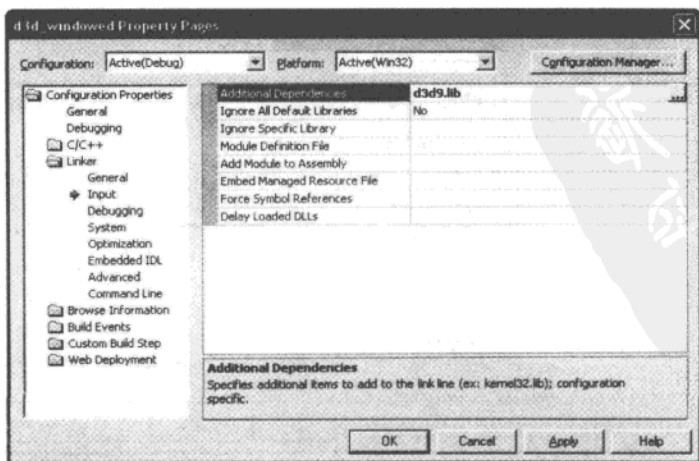


图 3-3 在 Additional Dependencies 字段中添加 d3d9.lib

假设源代码正确，那么编译 Direct3D 程序所需的一切就备齐了。我们现在已经在 Visual C++ 中配置了第一个 DirectX 项目！这可不容易啊！你应该觉得自己有了长足的进步了，尤其是那些刚接触 C++ 语言的读者！

**建议** 对项目进行编译时遇到了麻烦？这可能是因为你没有告诉 Visual C++ DirectX SDK 的安装位置。配置 Visual C++ 让其支持 DirectX 的方法可参阅附录 A。

## 2. 键入源代码

以下是让程序运行起来所需的标准 Windows 代码。在这一代码清单的末尾将给出 Direct3D 的特定代码。

```
/*
    Beginning Game Programming, Third Edition
    Chapter 3
    Direc3D_Windowed program
*/

#include <windows.h>
#include <d3d9.h>
#include <time.h>
#include <iostream>
using namespace std;

#pragma comment(lib,"d3d9.lib")
#pragma comment(lib,"d3dx9.lib")

//program settings
const string APPTITLE = "Direct3D_Windowed";
const int SCREENW = 1024;
const int SCREENH = 768;

//Direct3D objects
LPDIRECT3D9 d3d = NULL;
LPDIRECT3DDEVICE9 d3ddev = NULL;

bool gameover = false;

//macro to detect key presses
#define KEY_DOWN(vk_code) ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)

/**
** Game initialization function
**/
bool Game_Init(HWND window)
{
    MessageBox(window, "Game_Init", "BREAKPOINT", 0);

    //initialize Direct3D
    d3d = Direct3DCreate9(D3D_SDK_VERSION);
    if (d3d == NULL)
    {
```



```
        MessageBox(window, "Error initializing Direct3D", "Error", MB_OK);
        return 0;
    }

    //set Direct3D presentation parameters
    D3DPRESENT_PARAMETERS d3dpp;
    ZeroMemory(&d3dpp, sizeof(d3dpp));
    d3dpp.Windowed = TRUE;
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
    d3dpp.BackBufferFormat = D3DFMT_X8R8G8B8;
    d3dpp.BackBufferCount = 1;
    d3dpp.BackBufferWidth = SCREENW;
    d3dpp.BackBufferHeight = SCREENH;
    d3dpp.hDeviceWindow = window;

    //create Direct3D device
    d3d->CreateDevice(
        D3DADAPTER_DEFAULT,
        D3DDEVTYPE_HAL,
        window,
        D3DCREATE_SOFTWARE_VERTEXPROCESSING,
        &d3dpp,
        &d3ddev);

    if (d3ddev == NULL)
    {
        MessageBox(window, "Error creating Direct3D device", "Error", MB_OK);
        return 0;
    }

    return true;
}

/**
** Game update function
**/void Game_Run(HWND hwnd)
{
    //make sure the Direct3D device is valid
    if (!d3ddev) return;

    //clear the backbuffer to bright green
    d3ddev->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,255,0), 1.0f, 0);

    //start rendering
    if (d3ddev->BeginScene())
    {
        //do something?

        //stop rendering
        d3ddev->EndScene();

        //copy back buffer on the screen
        d3ddev->Present(NULL, NULL, NULL, NULL);
    }
}
```

```

    //check for escape key (to exit program)
    if (KEY_DOWN(VK_ESCAPE))
        PostMessage(hwnd, WM_DESTROY, 0, 0);
}

/**
** Game shutdown function
**/void Game_End(HWND hwnd)
{
    //display close message
    MessageBox(hwnd, "Program is about to end", "Game_End", MB_OK);

    //free memory
    if (d3ddev) d3ddev->Release();
    if (d3d) d3d->Release();
}

/**
** Windows event handling function
**/
LRESULT WINAPI WinProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam )
{
    switch( msg )
    {
        case WM_DESTROY:
            gameover = true;
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc( hWnd, msg, wParam, lParam );
}

/**
** Main Windows entry function
**/
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    //set the new window's properties
    //previously found in the MyRegisterClass function
    WNDCLASSEX wc;
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC)WinProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = NULL;
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = APPTITLE.c_str();
    wc.hIconSm = NULL;
    RegisterClassEx(&wc);
}

```



```
//create a new window
//previously found in the InitInstance function
HWND window = CreateWindow( APPTITLE.c_str(), APPTITLE.c_str(),
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, CW_USEDEFAULT,
    SCREENW, SCREENH,
    NULL, NULL, hInstance, NULL);

//was there an error creating the window?
if (window == 0) return 0;

//display the window
ShowWindow(window, nCmdShow);
UpdateWindow(window);

//initialize the game
if (!Game_Init(window)) return 0;

// main message loop
MSG message;
while (!gameover)
{
    if (PeekMessage(&message, NULL, 0, 0, PM_REMOVE))
    {
        TranslateMessage(&message);
        DispatchMessage(&message);
    }

    Game_Run(window);
}
Game_End(window);

return message.wParam;
}
```

关于这段代码，我们首先会注意到的是 `MyRegisterClass` 和 `InitInstance` 函数消失了！这些代码直接移到了 `WinMain` 中，因为 `Direct3D` 代码需要访问窗口句柄，而我们宁可选择把 `CreateWindow` 函数保留在 `WinMain` 里面。

在这段代码清单中还有许多更改，这与在上一章所看到的 `GameLoop` 程序有了区别。首先，当 `gameover` 变量为真时（无论是因为 `WM_DESTROY` 消息还是按了 `Esc` 键），`Game_End` 会被调用。这个函数在程序结束之前从内存中移除 `Direct3D` 对象。如果想看到程序挂起，只需直接终止程序而不释放 `Direct3D` 即可——它将在内存中继续运行，虽然程序窗口已经消失！这会是我们所说的“坏事一桩”。哦，为什么不直截了当地说呢？这实在是一个非常糟糕的事情。

现在，我们来看看对 `Direct3D` 进行初始化的代码。到目前为止，在本章所学过的代码都放在 `Game_Init` 之内，`WinMain` 会在主循环开始运行之前调用这个函数。对 `MessageBox` 的调用是为了做测试，一旦理解了程序的工作原理就可移除它。

```

int Game_Init(HWND hwnd)
{
    MessageBox(window, "Game_Init", "BREAKPOINT", 0);

    //initialize Direct3D
    d3d = Direct3DCreate9(D3D_SDK_VERSION);
    if (d3d == NULL)
    {
        MessageBox(window, "Error initializing Direct3D", "Error", MB_OK);
        return 0;
    }

    //set Direct3D presentation parameters
    D3DPRESENT_PARAMETERS d3dpp;
    ZeroMemory(&d3dpp, sizeof(d3dpp));
    d3dpp.Windowed = TRUE;
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
    d3dpp.BackBufferFormat = D3DFMT_X8R8G8B8;
    d3dpp.BackBufferCount = 1;
    d3dpp.BackBufferWidth = SCREENW;
    d3dpp.BackBufferHeight = SCREENH;
    d3dpp.hDeviceWindow = window;

    //create Direct3D device
    d3d->CreateDevice(
        D3DADAPTER_DEFAULT,
        D3DDEVTYPE_HAL,
        window,
        D3DCREATE_SOFTWARE_VERTEXPROCESSING,
        &d3dpp,
        &d3ddev);

    if (d3ddev == NULL)
    {
        MessageBox(window, "Error creating Direct3D device", "Error", MB_OK);
        return 0;
    }

    return true;
}

```

是否看到了调用 MessageBox 显示消息的第一行代码？插入这行代码是为了演示程序的工作原理、函数的调用方式并且演示 Windows 程序中事件的顺序。如果真想知道程序中所有的一切是如何工作的，那么可以在程序的每个地方都插入相似的 MessageBox 函数调用。基本上这个函数可以插入到除了游戏循环以外的任何地方，我们不想让消息框来中断游戏循环，因为如此的话会把一切都搞糟。好了，我们看一看 Game\_Run，了解在 Direct3D 显示器上绘制东西时会发生什么：

```

void Game_Run(HWND hwnd)
{
    //make sure the Direct3D device is valid
    if (!d3ddev) return;
}

```

```

//clear the backbuffer to bright green
d3ddev->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,255,0), 1.0f, 0);

//start rendering
if (d3ddev->BeginScene())
{
    //do something?
    //stop rendering
    d3ddev->EndScene();
    //copy back buffer on the screen
    d3ddev->Present(NULL, NULL, NULL, NULL);
}

//check for escape key (to exit program)
if (KEY_DOWN(VK_ESCAPE))
    PostMessage(hwnd, WM_DESTROY, 0, 0);
}

```

**建议** 在后面的章节中我们将花费大量的时间来讲解 Game\_Run 中的 Direct3D 渲染代码。目前由于我们还在学习 Direct3D 初始化，所以就将这些讲解放到以后再说。

首先，这个函数确认 d3ddev (Direct3D 设备) 存在；否则它返回一个错误。然后，调用 Clear 函数清除后台缓冲区——清除为绿色。对 Clear 的调用不只是为了装点门面，它会在对每个帧进行渲染之前确实地将屏幕清空（以后还会学到，这个函数也清除用于绘制多边形的 z 缓冲区）。想象一下我们让一个角色在屏幕上行走。每一帧（在这里是在 Game\_Run 中）我们都会更改成动画的下一帧，于是随着时间的推移这个角色看起来就真的像在行走。而如果一开始不清除屏幕，那么动画的每一帧就会绘制在前一帧之上，这样在屏幕上就会一团糟。这也是为什么在渲染开始之前调用 Clear 的原因，把以前的擦干净好为下一帧做准备。

再来看程序的最后一部分：

```

void Game_End(HWND hwnd)
{
    //display close message
    MessageBox(hwnd, "Game_End", "BREAKPOINT", MB_OK);

    //free memory
    if (d3ddev) d3ddev->Release();
    if (d3d) d3d->Release();
}

```

当 WM\_DESTROY 消息到来时，在 WinMain 中调用了 Game\_End 函数，你应该还记得吧。这通常发生在用户关闭程序时（单击右上角的小 X 图标）。

### 3. 运行程序

如果运行程序（在 Visual C++ 中按 F5 键），应该看到一个空白窗口弹出，如图 3-4 所示。嘿，它虽然没做什么，但我们学到了许多关于初始化 Direct3D 的知识——它已经可以弄出些多边形来了！

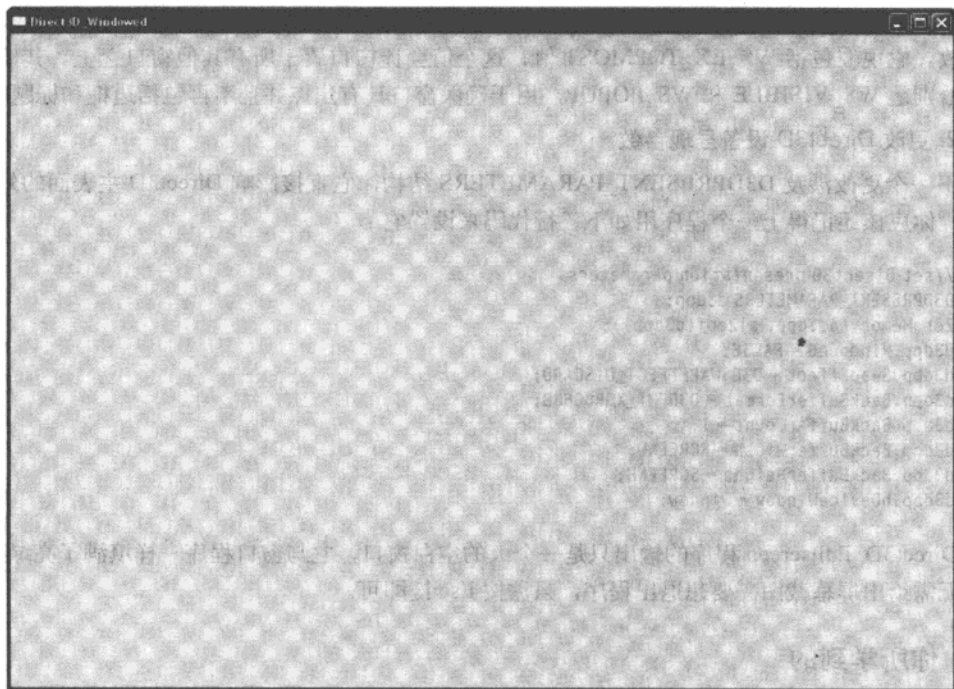


图 3-4 演示如何初始化 Direct3D 的 Direct3D\_Windowed 程序

### 3.1.4 全屏模式的 Direct3D

接下来要学习如何让 Direct3D 运行于全屏模式，这个模式是大多数游戏的运行模式。需要对 CreateWindow 函数调用及 Direct3D 呈现参数做些更改。以 Direct3D\_Windowed 程序作为基础，我们可以通过如下一些更改来让程序以全屏模式运行。

**提示** 作为产品，游戏最好运行于全屏模式。但在编程阶段让它运行于窗口模式也是可以的，因为全屏模式下的 Direct3D 会完全控制屏幕，我们无法看到弹出的错误消息。

#### 1. 修改 CreateWindow

现在，对 WinMain 中的 CreateWindow 函数调用做一些更改，你应该注意到了（这些更改以粗体表示）：

```
//create a new window
HWND window = CreateWindow(APPTITLE.c_str(), APPTITLE.c_str(),
    WS_EX_TOPMOST | WS_VISIBLE | WS_POPUP,
    CW_USEDEFAULT, CW_USEDEFAULT,
    SCREENW, SCREENH,
    NULL, NULL, hInstance, NULL);
```

CreateWindow 函数包括屏幕宽度和高度值，不过还对 WS\_OVERLAPPED 窗口样式做了一些更改。它现在包括 WS\_EX\_TOPMOST 值，这个值会让窗口置于所有其他窗口之上。其他两个选项分别是 WS\_VISIBLE 和 WS\_POPUP，用于确保窗口具有焦点并且不再包括边框和标题栏。

## 2. 更改 Direct3D 设备呈现参数

下一个更改涉及 D3DPRESENT\_PARAMETERS 结构，它直接影响 Direct3D 主表面的外观和容量。你应该还记得上一个程序用如下三行代码来设置它：

```
//set Direct3D presentation parameters
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory(&d3dpp, sizeof(d3dpp));
d3dpp.Windowed = FALSE;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dpp.BackBufferFormat = D3DFMT_X8R8G8B8;
d3dpp.BackBufferCount = 1;
d3dpp.BackBufferWidth = SCREENW;
d3dpp.BackBufferHeight = SCREENH;
d3dpp.hDeviceWindow = window;
```

Direct3D\_Fullscreen 程序的输出只是一个大的空白窗口，它与窗口程序一样填满了亮绿颜色，所以无需给出屏幕截图。要想退出程序，只需按 Esc 键即可。

## 3.2 你所学到的

本章我们学习了初始化及运行窗口模式和全屏模式下的 Direct3D 程序的方法。有以下几个要点：

- 学习了 Direct3D 接口对象。
- 学习了 CreateDevice 函数。
- 学习了 Direct3D 呈现参数。
- 学习了在窗口模式下运行 Direct3D 所需的设置。
- 学习了如何在全屏模式下运行 Direct3D。

## 3.3 复习测验

以下是一些复习测验题，可挑战你的过目不忘的能力，看你是否存在弱点。这些问题的答案可在附录 C 中找到。

- 1) Direct3D 是什么？
- 2) Direct3D 接口对象的名称是什么？
- 3) Direct3D 设备叫什么？
- 4) 用于启动渲染的 Direct3D 函数是哪个？
- 5) 可异步读入键盘的函数是哪个？
- 6) 主 Windows 函数——也就是以程序的“进入点”著称的函数，其名称是什么？
- 7) 在 Windows 程序中用于事件处理的函数，其常用名称是什么？

- 8) 哪个 Direct3D 函数在渲染完成后通过将后台缓冲区复制到视频内存的帧缓冲区中刷新屏幕?
- 9) 本书所用的 DirectX 是哪个版本?
- 10) Direct3D 的头文件叫什么?

### 3.4 自己动手

这些练习将给你带来挑战, 让你学习更多与本章给出的主题有关的知识, 帮助你提高自己的独立实践能力。

习题 1 修改 Direct3D\_Windowed 程序, 让它在背景上显示不同于绿色的其他颜色。

习题 2 修改 Direct3D\_Fullscreen 程序, 让它使用其他的分辨率, 而不是  $1024 \times 768$ 。







## 第二部分 >>>

# 游戏编程工具箱

---

第二部分讲解游戏编程的基本概念，即，让最简单的游戏工作起来都会需要的概念。例如键盘输入、鼠标输入、精灵、动画、计时、碰撞检测和响应等概念。还会探讨其他重要的关键主题。在阅读这些章节的过程中，就可开始创建自己的游戏编程工具箱——一组可以让使用 C++ 和 DirectX 编写游戏更为简单的、可重用的数据类型和函数。随着对每个新主题的讲解，在一组可重用的源代码文件中就会加入新的数据类型和函数，你可以在任何新的游戏项目中加入这些代码然后重用它们——而这是创建你自己的游戏引擎的第一个步骤！以下是第 2 部分中的各章，这些章节是本书的重头戏：

- 第 4 章 绘制位图
- 第 5 章 从键盘、鼠标和控制器获得输入
- 第 6 章 绘制精灵并显示精灵动画
- 第 7 章 精灵变换
- 第 8 章 检测精灵碰撞
- 第 9 章 打印文本
- 第 10 章 卷动背景
- 第 11 章 播放音频
- 第 12 章 3D 渲染基础
- 第 13 章 渲染 3D 模型文件



## 第4章 绘制位图

史上最好的游戏中有一些是 2D 游戏，它们根本就不需要高级的 3D 加速视频卡。学习 2D 图形很重要，因为它是显示在显示器上的所有图形的基础——无论这些图形如何渲染。而且，游戏图形都需要转换成屏幕上的像素阵列。本章我们将学习表面，这是可以绘制在屏幕上的常规位图（regular bitmap）。好，回想一下你最爱不释手的那些游戏。它们都是 3D 游戏吗？很可能不是。牛气冲天的游戏中，2D 游戏的数量要比 3D 游戏多。与其在这里对比、比较 2D 和 3D，还不如都学习它们，然后按照游戏的需要来使用。游戏程序员应该知道创建最好游戏的所有事情。

本章将学到：

- 如何在内存中创建表面。
- 如何用颜色来填充表面。
- 如何装载位图图像文件。
- 如何将表面绘制在屏幕上。

### 4.1 表面和位图

我们从最简单的技术开始 Direct3D 图形编程之旅——使用 Direct3D 表面来装载并绘制位图。表面很容易操作，因为 Direct3D 在内部使用它们将图形发送到帧缓冲区，即视频卡实际将像素绘制到屏幕上的那个部分。虽然 Direct3D 表面易于使用，但它们还是有一些限制，例如不支持透明。在第 6 章（“绘制精灵并显示精灵动画”）之前，我们还没有学习纹理和精灵，所以我们将不得不用变通的手段绕过这个限制。当然 Direct3D 表面不是个短效的技术，不要在本章之后就抛弃表面。实际上，我们将在第 10 章（“卷动背景”）中使用表面来实现卷动。卷动是笔者最喜欢的主题之一！现在，我们继续学习如何使用 Direct3D 表面。

Direct3D 使用表面来处理许多事情。显示器（如图 4-1 所示）显示视频卡发送给它的内容，视频卡从帧缓冲区中取出视频显示内容并且一次一个像素地发送给显示器（它们可以处于单个文件中，不过它们移得相当快！）。

帧缓冲区位于视频卡本身的内存芯片中（如图 4-2 所示），这些芯片通常非常快。曾几何时，因为其速度太快——要比标准系统 RAM 快得多，所以视频内存（VRAM）极端昂贵。现在这个世界反过来了，PC 的主内存通常拥有最好的技术，而视频卡则落后一两步。个中原因是因为要重新架构视频卡较为困难，这是个非常精确而复杂的电路板。

**建议** Direct3D 表面在游戏项目中既不用于精灵也不用于纹理，但它却具备按位块传输（blit）的能力，这就是我们现在讲解它的原因。它与基于纹理的图形仅一步之遥。

另一方面，PC 主板永远处在不断改变的状态中，因为半导体公司之间相互竞争谁都不甘落后。而视频卡公司，无论它们有多大的竞争力，却不能将六个月的工作押在会被其他内存技术取

代的不被市场认可的内存技术上（还记得 Rambus 内存吧？Intel 曾经试图在 Pentium III 处理器的最早版本中支持它）。而且，由于主板是为不同的行业和用途而构建的，它们因此必须经历更多的试验检验；而视频卡的构建只有一个目的：显示图形。所以，视频卡上的芯片没有经过那么多试验。在 PC 市场对某个内存标准有了取向后，那么它就有出现在视频卡中的趋势。你也许会想起第一款 DDR（双数据率）内存存在视频卡上使用的情形，此时距离 DDR 最初发布已经过去有一段时间了。

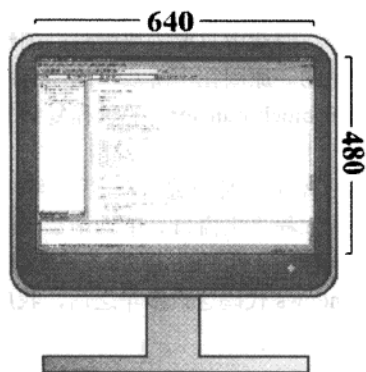


图 4-1 典型的显示器

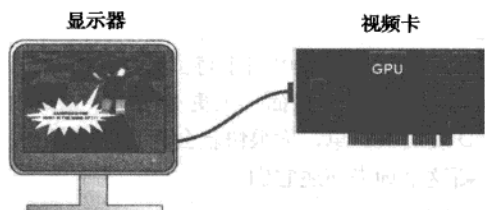


图 4-2 显示器显示由视频卡发送给它的线性像素阵列

帧缓冲区位于视频内存中，代表要在显示器上显示的图像（如图 4-3 所示）。所以，创建图形最简单的方法就是直接修改帧缓冲区，这是合情合理的，结果就是我们可以立即看到改变。基本上一切就是这样工作的，但这里漏掉了一个小细节。我们不会直接在帧缓冲区上绘图，因为在绘制、擦除、移动及重绘图形的同时屏幕正被刷新，这会导致抖动。我们要做的是，在一个离屏缓冲区上绘制所有的一切，然后将这个“双重”或者“后台”缓冲区非常快速地喷到屏幕上。这称为“双缓冲（double buffering）”。还有其他的创建不抖动的显示的方法，例如页翻转，但我更喜欢选择后台缓冲区（back buffer），因为它更为直白（而且容易一些）。

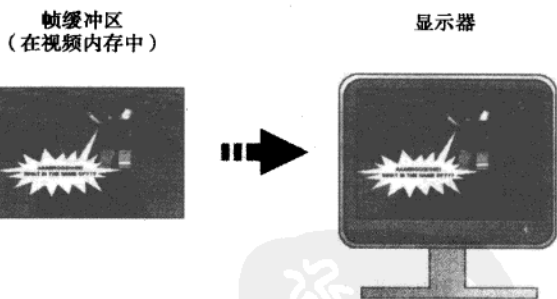


图 4-3 视频内存中的帧缓冲区包含在显示器上渲染的图像

#### 4.1.1 主表面

上一章我们通过设置呈现参数创建了一个后台缓冲区。而后，通过使用 Clear 函数，将后台缓冲区填满了绿色，并使用 Present 函数刷新屏幕。在不知不觉中，我们使用了双重/后台缓冲区！这是 Direct3D 提供的一个美妙的特性——一个内置的后台缓冲区。这是合理的，因为在今

天，在游戏中双缓冲就如厨房中面包和奶油一样普遍。

早先提及的“帧缓冲区”也称为前台缓冲区（front buffer），这种叫法是有道理的，因为后台缓冲区的每一帧内容都会被复制到它上面。在配置呈现参数并调用 CreateDevice 时，前台和后台缓冲区都已创建完成。难道这还不够棒吗？

#### 4.1.2 从离屏（off-screen）表面

从表面或者离屏表面是我们可使用的另外一种表面。这种类型的表面实际上只是在内存中看起来像个位图（它有个头部，然后是代表像素的数据）的数组。在游戏中只要需要就可以创建任意多个离屏表面，在游戏运行中使用上百个表面和纹理都是常见的。原因在于游戏中的所有图形都储存在表面或纹理中，这些图像都通过一个名为位块传输（bit-block transfer）的过程复制到屏幕上。我们通常使用“blitter”这个词来代表这个术语。

记得在第2章的 GameLoop 程序中使用了名为 BitBlt 的函数（那时有意忽略不讲解它）。BitBlt 是 Windows GDI 中用于将位图“blitting”到设备环境（例如程序主窗口）中的函数。设备环境如同 Direct3D 的表面，但更难使用（这是由 Windows GDI 的复杂性所决定的）。相比之下，Direct3D 表面很简单，你很快就会看到。实际上，在编写了 Windows 代码这么多年之后，我应该使用刷新这个词来描述它们。

##### 1. 创建表面

要创建 Direct3D 表面，首先要声明一个指向内存中的表面的变量。表面对象名称为 LPDIRECT3DSURFACE9，如下创建这个变量：

```
LPDIRECT3DSURFACE9 surface = NULL;
```

一旦创建了表面，就可以对其做许多操作了。可以使用位块传输（StretchRect）将位图绘制到表面上（当然得从其他表面），也可用颜色来填充表面，等等。例如，如果想在绘制之前清除表面上的内容，那么可以使用 ColorFill 函数，其语法如下：

```
HRESULT ColorFill(
    IDirect3DSurface9 *pSurface,
    CONST RECT *pRect,
    D3DCOLOR color
);
```

以下代码将目标表面用红颜色填充：

```
d3ddev->ColorFill(surface, NULL, D3DCOLOR_XRGB(255,0,0));
```

##### 2. 绘制表面（Blitting）

Blitter 可能是最有趣的函数。可以将某个表面的一部分或全部位块传输到另一个表面（包括屏幕上的后台缓冲区）。这个 blitter 称为 StretchRect（奇怪的名称，是吧？）。它的定义如下所示：

```

HRESULT StretchRect(
    IDirect3DSurface9 *pSourceSurface,
    CONST RECT *pSourceRect,
    IDirect3DSurface9 *pDestSurface,
    CONST RECT *pDestRect,
    D3DTEXTUREFILTERTYPE Filter
);

```

嗯，难道我没告诉你 Direct3D 要比 Windows GDI 更容易处理位图吗？我可不是在开玩笑。这个小函数只有 5 个参数，它真是很容易使用。我们来看个例子：

```
d3ddev->StretchRect(surface, NULL, backbuffer, NULL, D3DTEXF_NONE);
```

这是调用该函数最简单的方法——假设两个表面尺寸一致。如果源表面比目标表面小，那么就会传输到目标表面的左上角。当然，这算不上什么，这个函数真正方便之处是在我们为源和目标表面指定矩形区域时。源矩形可以是整个表面的一小部分，目标表面也是如此，而我们通常将源传输到目标“上面”的某个位置上。以下是一个示例：

```

rect.left = 100;
rect.top = 90;
rect.right = 200;
rect.bottom = 180;
d3ddev->StretchRect(surface, NULL, backbuffer, &rect, D3DTEXF_NONE);

```

这段代码将源表面复制到目标，将其伸展到 (100, 90, 200, 180) 位置上的矩形，一共是 100 像素 × 90 像素。无论源表面的尺寸多大，只要不是 NULL，都可以“填塞”到目标矩形的范围内。

我在没有事先讲解 backbuffer 的来历的情况下就使用了它。不，没有一个名为 backbuffer 可以自由使用的全局变量（虽然这会很酷）！但这不是什么大事，我们可以自己创建这个变量。它通常只是指向真实后台缓冲区的指针，通过调用 GetBackBuffer 这个特殊函数可以得到这个指针。好家伙，这个调用够费力的。但这是个直白的方法（这不是 Microsoft 的通常方法），这一点没什么好争论的。

```

HRESULT GetBackBuffer(
    UINT iSwapChain,
    UINT BackBuffer,
    D3DBACKBUFFER_TYPE Type,
    IDirect3DSurface9 **ppBackBuffer
);

```

以下是调用这个函数来获取指向后台缓冲区指针的方法。首先，创建一个 backbuffer 变量（也就是一个指针），然后让神奇的 GetBackBuffer 函数将它“指向”真正的后台缓冲区：

```

LPDIRECT3DSURFACE9 backbuffer = NULL;
d3ddev->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, &backbuffer);

```

我敢肯定你会担心 Direct3D 能否做得到。嗯，这是你自己的观点。对 DirectX SDK 帮助文件（我经常参考它）中的每个未知函数抱以悲观和抱怨是可以的，但也可以学以致用、如法炮制，

并且开始编写游戏。我们肯定是要绘制多边形的，很快就会学到。

### 4.1.3 Create\_Surface 示例

把这一切转变为示例程序，这样可以很好地了解它们。这里编写了一个能够演示 ColorFill、StretchRect 和 GetBackBuffer，而且更重要的是，它是个能够演示如何使用表面的程序，名为 Create\_Surface。可以从图 4-4 看到示例输出。你可能疑惑为什么图中不是只有一个矩形，那是因为程序在运行的同一时间只有一个矩形在屏幕上，但它运行的速度很快，以至于我们会觉得屏幕上同时有许多矩形。

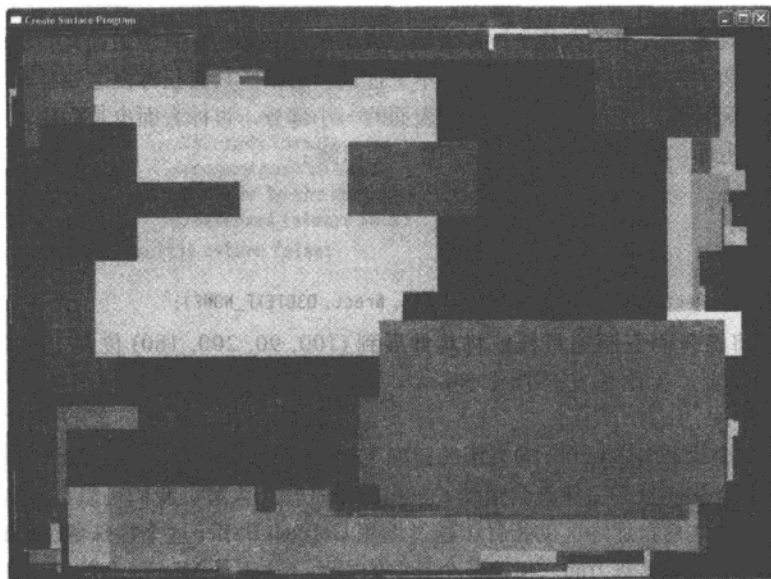


图 4-4 Create\_Surface 程序从离屏表面将随机矩形复制到屏幕上

然后，创建一个新的名为 Create\_Surface 的项目并且将名为 winmain.cpp 的新文件添加到项目中。如以前那样，进入 Project 菜单，单击 Settings，再单击 Linker/Input 项，然后将 d3d9.lib 添加到 Additional Dependencies 字段中。

准备好了吗？好，我们开始吧。以下是程序代码。关键代码行以粗体方式显示，如果通过修改第 3 章的示例程序（本程序原本的基础）来输入代码，则可以很容易找到它们。

**建议** 可以从 CD-ROM 中装载这一项目；或者对第 3 章的程序进行修改，因为大多数 Windows 代码都一样。

```
/*
Beginning Game Programming, Third Edition
Chapter 4
Create_Surface program
*/
```

```

#include <windows.h>
#include <d3d9.h>
#include <time.h>
#include <iostream>
using namespace std;

//application title
const string APPTITLE = "Create Surface Program";

//macro to read the keyboard
#define KEY_DOWN(vk_code) ((GetAsyncKeyState(vk_code)&0x8000)?1:0)

//screen resolution
#define SCREENW 1024
#define SCREENH 768

//Direct3D objects
LPDIRECT3D9 d3d = NULL;
LPDIRECT3DDEVICE9 d3ddev = NULL;
LPDIRECT3DSURFACE9 backbuffer = NULL;
LPDIRECT3DSURFACE9 surface = NULL;

bool gameover = false;

/**
** Game initialization function
**/
bool Game_Init(HWND hwnd)
{
    //initialize Direct3D
    d3d = Direct3DCreate9(D3D_SDK_VERSION);
    if (d3d == NULL)
    {
        MessageBox(hwnd, "Error initializing Direct3D", "Error", MB_OK);
        return false;
    }

    //set Direct3D presentation parameters
    D3DPRESENT_PARAMETERS d3dpp;
    ZeroMemory(&d3dpp, sizeof(d3dpp));
    d3dpp.Windowed = TRUE;
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
    d3dpp.BackBufferFormat = D3DFMT_X8R8G8B8;
    d3dpp.BackBufferCount = 1;
    d3dpp.BackBufferWidth = SCREENW;
    d3dpp.BackBufferHeight = SCREENH;
    d3dpp.hDeviceWindow = hwnd;

    //create Direct3D device
    d3d->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hwnd,
        D3DCREATE_SOFTWARE_VERTEXPROCESSING, &d3dpp, &d3ddev);

    if (!d3ddev)
    {

```



```

    MessageBox(hwnd, "Error creating Direct3D device", "Error", MB_OK);
    return false;
}

//set random number seed
srand(time(NULL));

//clear the backbuffer to black
d3ddev->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,0), 1.0f, 0);

//create pointer to the back buffer
d3ddev->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, &backbuffer);

//create surface
HRESULT result = d3ddev->CreateOffscreenPlainSurface(
    100,                //width of the surface
    100,                //height of the surface
    D3DFMT_X8R8G8B8,    //surface format
    D3DPPOOL_DEFAULT,   //memory pool to use
    &surface,            //pointer to the surface
    NULL);              //reserved (always NULL)
if (!SUCCEEDED(result)) return false;

return true;
}

/**
** Game update function
**/
void Game_Run(HWND hwnd)
{
    //make sure the Direct3D device is valid
    if (!d3ddev) return;

    //start rendering
    if (d3ddev->BeginScene())
    {
        //fill the surface with random color
        int r = rand() % 255;
        int g = rand() % 255;
        int b = rand() % 255;
        d3ddev->ColorFill(surface, NULL, D3DCOLOR_XRGB(r,g,b));

        //copy the surface to the backbuffer
        RECT rect;
        rect.left = rand() % SCREENW/2;
        rect.right = rect.left + rand() % SCREENW/2;
        rect.top = rand() % SCREENH;
        rect.bottom = rect.top + rand() % SCREENH/2;
        d3ddev->StretchRect(surface, NULL, backbuffer, &rect, D3DTEXF_NONE);

        //stop rendering
        d3ddev->EndScene();
    }
}

```

```

        //display the back buffer on the screen
        d3ddev->Present(NULL, NULL, NULL, NULL);
    }

    //check for escape key (to exit program)
    if (KEY_DOWN(VK_ESCAPE))
        PostMessage(hwnd, WM_DESTROY, 0, 0);

}

/**
** Game shutdown function
**/
void Game_End(HWND hwnd)
{
    if (surface) surface->Release();
    if (d3ddev) d3ddev->Release();
    if (d3d) d3d->Release();
}

/**
** Windows event callback function
**/

LRESULT WINAPI WinProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam )
{
    switch( msg )
    {
        case WM_DESTROY:
            gameover = true;
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc( hWnd, msg, wParam, lParam );
}

/**
** Windows entry point function
**/
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    //create the window class structure
    WNDCLASSEX wc;
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC)WinProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = NULL;
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);

```

```

wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
wc.lpszMenuName = NULL;
wc.lpszClassName = APPTITLE.c_str();
wc.hIconSm = NULL;
RegisterClassEx(&wc);

//create a new window
HWND window = CreateWindow(APPTITLE.c_str(), APPTITLE.c_str(),
    WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
    SCREENW, SCREENH, NULL, NULL, hInstance, NULL);

//was there an error creating the window?
if (window == 0) return 0;

//display the window
ShowWindow(window, nCmdShow);
UpdateWindow(window);

//initialize the game
if (!Game_Init(window)) return 0;

// main message loop
MSG message;
while (!gameover)
{
    if (PeekMessage(&message, NULL, 0, 0, PM_REMOVE))
    {
        TranslateMessage(&message);
        DispatchMessage(&message);
    }

    Game_Run(window);
}

return message.wParam;
}

```

#### 4.1.4 从磁盘装载位图

在了解了使用 StretchRect 函数绘制位图的方法之后，可以进入下一个步骤——从文件中将位图装载到表面对象中，并将它从这里绘制出来。遗憾的是，Direct3D 没有任何用于装载位图文件的函数，所以我们不得不编写自己的位图装载器。

实际上，我在此刻想到了电视剧《Perfect Stranger》中的 Balki Bartokamous，他有一句名言：“别那么滑稽！”为运行于 Windows 中的程序编写自己的位图装载器，确实很滑稽。因为 Windows 是 BMP 文件的本地平台啊！

然而，事实是这样的，Direct3D 真的不知道如何装载位图。幸运的是，我们有名为 D3DX（代表 Direct3D Extensions——Direct3D 扩展）的助手库，它提供了许多有帮助的函数，包括一个将位图装载到表面的函数。唯一的条件就是需要在程序中加一行 #include <d3dx.h> 语句，并且将 d3dx9.lib 添加到项目设置中。这不是什么难事。

**建议** 为什么 Microsoft 里的人只要不能给新产品想个好名字，就把它叫做“X”呢？DirectX、Windows XP、XNA Game Studio、Xbox、Xbox 360、Office XP。“X”是 20 世纪 90 年代的潮流，但潮得过了。曾经有一款名为 Microsoft Bob 的软件，但卖得不好。这也许是件好事，否则我们就会看到这家公司的很多标着“Bob”的产品了。

这里我们感兴趣的函数是 D3DXLoadSurfaceFromFile，语法如下：

```
HRESULT D3DXLoadSurfaceFromFile(
    LPDIRECT3DSURFACE9 pDestSurface,
    CONST PALETTEENTRY* pDestPalette,
    CONST RECT* pDestRect,
    LPCTSTR pSrcFile,
    CONST RECT* pSrcRect,
    DWORD Filter,
    D3DCOLOR ColorKey,
    D3DXIMAGE_INFO* pSrcInfo
);
```

好了，现在来看个让人愉悦的东西。这个伟大的函数不仅可以装载标准 Windows 位图文件，而且还可以装载一大堆其他格式的文件！表 4-1 列出了这些格式。

表 4-1 图形文件格式

扩展名	格 式
.bmp	Windows 位图
.dds	DirectDraw 表面
.dib	Windows 设备无关位图
.jpg	联合图像专家组 (JPEG)
.png	可移植网络图形
.tga	Truevision Targa

这些参数有许多通常都是 NULL，所以它并不像它看起来那么难用（虽然在我看到超过 6 个参数的函数时，我的眼睛就一片昏花）。

#### 4.1.5 Load\_Bitmap 程序

下面写一个小程序来演示将位图文件装载到表面并绘制到屏幕上的方法。首先，无需再一次键入所有的代码，只需对 Create\_Surface 程序按提及的更改进行修改即可。所以这里只列出进行这些修改所需的代码。其次，需要演示如何为 D3DX 配置项目。打开 Project Settings 对话框，单击 Link 选项卡，将 d3d9.lib 和 d3dx9.lib 都输入到 Additional Dependencies 字段，如图 4-5 所示。是否还记得另外一种将库文件添加到项目中的方法？

首先要做的是将 #include <d3dx9.h> 语句添加到代码中。如下所示：

```

/*
    Beginning Game Programming, Third Edition
    Chapter 4
    Load Bitmap program
*/
#include <windows.h>
#include <d3d9.h>
#include <d3dx9.h>
#include <time.h>
#include <iostream>
using namespace std;
//program values
const string APPTITLE = "Load Bitmap Program";
const int SCREENW = 1024;
const int SCREENH = 768;

```

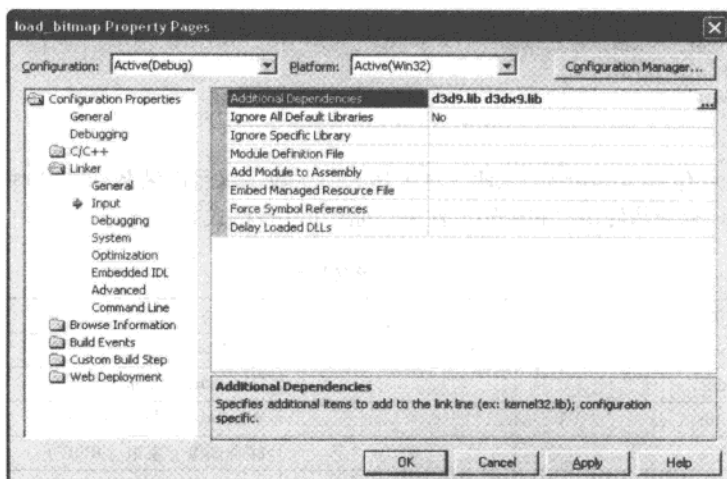


图 4-5 将对 D3DX 库的支持添加到项目中

现在，向下滚动代码到 `Game_Init` 函数，按照粗体的内容进行更改（删除上一个项目中不再需要的代码行）。大多数代码都保留原样。

```

/**
** Game initialization function
**/
bool Game_Init(HWND window)
{
    //initialize Direct3D
    d3d = Direct3DCreate9(D3D_SDK_VERSION);
    if (!d3d)
    {
        MessageBox(window, "Error initializing Direct3D", "Error", MB_OK);
        return false;
    }
}

```

```

//set Direct3D presentation parameters
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory(&d3dpp, sizeof(d3dpp));
d3dpp.Windowed = TRUE;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
d3dpp.BackBufferFormat = D3DFMT_X8R8G8B8;
d3dpp.BackBufferCount = 1;
d3dpp.BackBufferWidth = SCREENW;
d3dpp.BackBufferHeight = SCREENH;
d3dpp.hDeviceWindow = window;

//create Direct3D device
d3d->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, window,
    D3DCREATE_SOFTWARE_VERTEXPROCESSING, &d3dpp, &d3ddev);

if (!d3ddev)
{
    MessageBox(window, "Error creating Direct3D device", "Error", MB_OK);
    return 0;
}

//clear the backbuffer to black
d3ddev->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,0), 1.0f, 0);

//create surface
HRESULT result = d3ddev->CreateOffscreenPlainSurface(
    SCREENW,           //width of the surface
    SCREENH,           //height of the surface
    D3DFMT_X8R8G8B8,   //surface format
    D3DPPOOL_DEFAULT,  //memory pool to use
    &surface,           //pointer to the surface
    NULL);              //reserved (always NULL)

if (!SUCCEEDED(result)) return false;

//load surface from file into newly created surface
result = D3DXLoadSurfaceFromFile(
    surface,            //destination surface
    NULL,               //destination palette
    NULL,               //destination rectangle
    "legotron.bmp",     //source filename
    NULL,               //source rectangle
    D3DX_DEFAULT,       //controls how image is filtered
    0,                  //for transparency (0 for none)
    NULL);              //source image info (usually NULL)

//make sure file was loaded okay
if (!SUCCEEDED(result)) return false;

return true;
}

```

需要对 Game\_Run 做一些更改，主要移除一些代码，因为在图像绘制之后不会有屏幕更新发生。

```

/**
** Game update function
**/
void Game_Run(HWND hwnd)
{
    //make sure the Direct3D device is valid
    if (d3ddev) return;

    //create pointer to the back buffer
    d3ddev->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, &backbuffer);

    //start rendering
    if (d3ddev->BeginScene())
    {
        //draw surface to the backbuffer
        d3ddev->StretchRect(surface, NULL, backbuffer, NULL, D3DTEXF_NONE);

        //stop rendering
        d3ddev->EndScene();
        d3ddev->Present(NULL, NULL, NULL, NULL);
    }

    //check for escape key (to exit program)
    if (KEY_DOWN(VK_ESCAPE))
        PostMessage(hwnd, WM_DESTROY, 0, 0);
}

```

本程序完整的源代码清单和项目包含在 CD-ROM 的 \sources\chapter04\Load Bitmap Demo 文件夹中。运行本程序时，应该能看到图 4-6 所示的位图填充了屏幕。



图 4-6 Load\_Bitmap 程序将一个位图图像装载到 Direct3D 表面，然后将其传输到屏幕

### 4.1.6 代码再利用

你是否注意到在最近两章中 Windows 代码一点儿都没改变？WinProc 和 WinMain 在各个程序中都是一样的。我们已经看到每个新项目中代码回收利用的好处了。通过将程序中不一致的部分移到标准的游戏函数（Game\_Init、Game\_Run 和 Game\_End）中，可以对调用函数进行再利用，这样 WinProc 和 WinMain 将无需包含在每个项目中。当然，这些函数是需要存在于程序中的，CD-ROM 中的所有示例都完整地带有这些函数，但我们无需关注它们并且在每一章都重复这些代码清单。少打字对我们的心智有好处；而少用纸张则是在保护环境！多好啊！还有，这也解释了为什么一本这么复杂的书只有这么一点篇幅——因为我们编写紧凑的代码！

## 4.2 你所学到的

本章我们学习了如何创建并操纵表面。有以下几个要点：

- 如何创建表面。
- 可以使用随机的颜色来填充表面。
- 将位图图像从磁盘装载到表面的方法，这种方法支持多种图形文件格式。
- 将整个或部分表面绘制到屏幕的方法。

## 4.3 复习测验

以下是一些复习测验题，以巩固所学内容。附录 C 给出了这些习题答案。

- 1) 主 Direct3D 对象的名称是什么？
- 2) Direct3D 设备的名称是什么？
- 3) Direct3D 表面对象的名称是什么？
- 4) 用于将 Direct3D 表面绘制到屏幕上的是什么函数？
- 5) 描述复制内存中的图像的术语是什么？
- 6) 用于处理 Direct3D 表面的结构的名称是什么？
- 7) 同一个结构的长指针定义版本的名称是什么？
- 8) 返回 Direct3D 后台缓冲区指针的是哪个函数？
- 9) 哪个 Direct3D 设备函数用给定颜色填充表面？
- 10) 哪个函数用于将位图文件装载到内存中的 Direct3D 表面？

## 4.4 自己动手

这些练习将帮助大家巩固今天所学的内容。虽然作用有限，但值得一试。

习题 1 Load\_Bitmap 程序装载位图文件并在屏幕上显示。使用所学的关于 StretchRect 的知识只将位图图像的一部分绘制到屏幕上。

习题 2 星际联盟招募了你上前线防卫来自扎克行星的攻击。使用本章所学知识编写一个简单的程序以显示你学有所用。



## 第 5 章 从键盘、鼠标和控制器获得输入

欢迎来到虚拟接口的一章！接下来，我们将学习如何使用 DirectInput 进行键盘和鼠标编程，从而为游戏提供对最常见的输入设备的支持。还将学习对 Xbox 360 控制器的编程方法（如果你有的话，我强烈推荐它，因为如果有它相关示例会更有趣！）。我们将用装载的位图创建简单的 2D 图像，每个都基于在第 4 章所学的 Direct3D 表面；本章我们将使用它来制作一个名为 Bomb Catcher 的游戏。所以，要提起精神！这个游戏有助于演示在游戏中如何同时使用键盘、鼠标及控制器。

本章将学到：

- 如何创建 DirectInput 设备。
- 如何从键盘获取输入。
- 如何从鼠标获取输入。
- 如何从 Xbox 360 控制器获取输入。
- 如何创建并移动一个精灵。
- 如何制作你的第一个游戏。

### 5.1 键盘输入

键盘是所有游戏的标准输入设备，甚至是那些并不特别适用键盘的游戏的标准输入设备。所以游戏或多或少会使用键盘是个事实。至少应该允许用户按 Esc 键退出游戏或进入某种形式的游戏内菜单（这是标准）。使用 DirectInput 对键盘编程不困难，但首先还是需要初始化 DirectInput。

主 DirectInput 设备称为 IDirectInput8，可以使用 LPDIRECTINPUT8 指针数据类型直接引用它。为什么在这些接口后带有“8”这个数字呢？因为，如 DirectSound 一样，DirectInput 很长时间没有改变。这会让人困惑，为什么说我们已经完整升级到 9.0c 版（当你读到这里时很可能已经有新版本了）。

DirectInput 库文件名为 dinput8.lib，要确认在 Project Settings 对话框的 Linker 选项卡中添加了 this 文件及其他库文件。这里假定读者阅读了第 4 章，而且，到此刻为止，学会了如何设置项目让其支持 DirectX 并且知道如何使用逐步建立起来的游戏框架。如果对如何设置项目有任何疑问，参阅第 4 章中完整的概述和指导。本章将在框架中使用两个新文件（dxinput.h 和 dxinput.cpp）添加一个新的组件用于 DirectInput。

#### 5.1.1 DirectInput 对象和设备

好了，我们对初始化 DirectX 组件已经熟悉了，那么就来学习如何扫描键盘来确定是否有按钮输入。首先定义程序要用的主 DirectInput 对象及设备的对象：

```
LPDIRECTINPUT8 dinput;  
LPDIRECTINPUTDEVICE8 dinputdev;
```

定义了变量之后，可调用 `DirectInputCreate8` 初始化 `DirectInput`。函数的格式如下：

```
HRESULT WINAPI DirectInput8Create(  
    HINSTANCE hinst,  
    DWORD dwVersion,  
    REFIID riidItf,  
    LPVOID *ppvOut,  
    LPUNKNOWN punkOuter  
);
```

这个函数创建传递给它的主 `DirectInput` 对象。第一个参数是当前程序的实例句柄。如果这个当前实例不能直接取得（通常只在 `WinMain` 中有），那么获得它的方便方法是用 `GetModuleHandle` 函数。第二个参数是 `DirectInput` 版本，总是传递在 `dinput.h` 中定义的 `DIRECTINPUT_VERSION` 即可。第三个参数是想要使用的 `DirectInput` 版本的引用标识符。目前，这个值是 `IID_IDirectInput8`。第四个参数是指向主 `DirectInput` 对象指针的指针（注意，这里是双重指针），而第五个参数总是 `NULL`。以下是调用这个函数的方法示例：

```
HRESULT result = DirectInput8Create(  
    GetModuleHandle(NULL),  
    DIRECTINPUT_VERSION,  
    IID_IDirectInput8,  
    (void**)&dinput,  
    NULL);
```

在初始化了对象之后，可通过调用 `CreateDevice` 函数使用这个对象创建新的 `DirectInput` 设备：

```
HRESULT CreateDevice(  
    REFGUID rguid,  
    LPDIRECTINPUTDEVICE *lplpDirectInputDevice,  
    LPUNKNOWN pUnkOuter  
);
```

第一个参数的值指定要创建的对象类型（例如键盘或鼠标）。这个参数可用的值如下：

- `GUID_SysKeyboard`
- `GUID_SysMouse`

第二个参数是接收 `DirectInput` 设备句柄的地址的设备指针。第三个参数总是 `NULL`。以下是调用这个函数的方法：

```
result = dinput->CreateDevice(GUID_SysKeyboard, &dikeyboard, NULL);
```

### 5.1.2 初始化键盘

一旦拥有了键盘的 `DirectInput` 对象及设备对象，就可以初始化键盘句柄，为输入做准备。下一步是设置键盘的数据格式，也就是告诉 `DirectInput` 如何将数据传回给程序。以这种方式进行抽象是因为市场上有上百种输入设备，各有不同，必须要有统一的能够读取它们的方式。

### 1. 设置数据格式

使用 `SetDataFormat` 指定数据格式。

```
HRESULT SetDataFormat(  
    LPCDATAFORMAT lpdf  
);
```

该函数唯一的参数指定设备类型。对于键盘，应传递 `c_dfDIKeyboard` 值作为参数。为鼠标准备的常量为 `c_dfDIMouse`。以下是一个简单的函数调用：

```
HRESULT result = dikeyboard->SetDataFormat(&c_dfDIKeyboard);
```

注意，不需要自己定义 `c_dfDIKeyboard`，它在 `dinput.h` 中定义。

### 2. 设置协作级别

下一步是设置协作级别，它按优先级决定 `DirectInput` 将键盘输入传递给程序的程度。要设置协作级别，可调用 `SetCooperativeLevel` 函数：

```
HRESULT SetCooperativeLevel(  
    HWND hwnd,  
    DWORD dwFlags  
);
```

第一个参数是窗口句柄。第二个是个有趣的参数，因为它指定程序对键盘或鼠标所拥有的优先级。在键盘上最常用的值是 `DISCL_NONEXCLUSIVE` 和 `DISCL_FOREGROUND`。如果想获得对键盘的排他使用，`DirectInput` 可能会报告，所以可以以前台应用程序优先级请求非排他访问，这样可以给游戏最多的键盘控制。如此一来，调用这个函数的方法就是：

```
HRESULT result = dikeyboard->SetCooperativeLevel(hwnd,  
    DISCL_NONEXCLUSIVE | DISCL_FOREGROUND);
```

### 3. 获取设备

初始化键盘的最后一步是使用 `Acquire` 函数获取键盘设备：

```
HRESULT Acquire(VOID);
```

如果函数返回正值 (`DI_OK`)，则说明成功地获得了键盘，此时可以开始检查键盘按键了。

在这里需要说明的重要一点是，在游戏结束之前必须反获取（还回）键盘，否则 `DirectInput` 和键盘句柄会处于未知状态。Windows 和 `DirectInput` 可能会为我们处理清理工作，但这依赖于用户所运行的 Windows 版本。信不信由你，现在还有运行 Windows 98 和 ME 的计算机，尽管这些操作系统都已经很落后了。虽然 Windows XP（及后来的版本）要稳定得多，只是我们不能任凭任何东西自生自灭。在游戏结束前最好还是反获取设备。每个 `DirectInput` 设备都有一个 `Unacquire` 函数，格式如下：

```
HRESULT Unacquire(VOID);
```

### 5.1.3 读取键盘按键

需要在游戏循环的某个地方轮询键盘以便更新键值。说到键，我们需要定义一个键的数组，以便接受键盘设备状态，如下所示：

```
char keys[256];
```

我们必须通过轮询键盘来填充这个字符数组，实现这个目标需要调用 `GetDeviceState` 函数。这个函数可用于所有的设备，无论设备类型是什么，所以对于所有的输入设备它是标准的：

```
HRESULT GetDeviceState(
    DWORD cbData,
    LPVOID lpvData
);
```

第一个参数是用于填充数据的设备状态缓冲区的大小。第二个参数是指向数据的指针。对于键盘，调用这个函数的方法是：

```
dkeyboard->GetDeviceState(sizeof(keys), (LPVOID)&keys);
```

在轮询了键盘之后，就可以检查 `keys` 数组中与 `DirectInput` 键码相关的值；这些值列在本书附录 B 中。

检查 `Esc` 键的方法如下所示：

```
if (keys[DIK_ESCAPE] & 0x80)
{
    //ESCAPE key was pressed, so do something!
}
```

## 5.2 鼠标输入

一旦编写了键盘处理器，那么加入对鼠标的支持就是小菜一碟了，因为它们的代码非常相似并且共享 `DirectInput` 对象和设备指针。所以我们就直接学习鼠标接口吧。首先，定义鼠标设备：

```
LPDIRECTINPUTDEVICE8 dimouse;
```

下一步，创建鼠标设备：

```
result = dinput->CreateDevice(GUID_SysMouse, &dimouse, NULL);
```

### 5.2.1 初始化鼠标

这里假定 `DirectInput` 已经准备就绪，现在想做的就是添加一个鼠标处理器。下一步是设置鼠标的格式，它告诉 `DirectInput` 如何将数据传回给程序。鼠标和键盘的工作方式完全一样。

#### 1. 设置数据格式

`SetDataFormat` 函数如下所示：

```
HRESULT SetDataFormat(
    LPCDI_DATAFORMAT lpdf
);
```

该函数的唯一参数指定设备类型。鼠标的常量是 `c_dfDIMouse`。然后，就是一个简单的函数调用：

```
HRESULT result = dimouse->SetDataFormat(&c_dfDIMouse);
```

再次注意，无需定义 `c_dfDIMouse`，因为它已定义在 `dinput.h` 中。

## 2. 设置协作级别

下一步是设置协作级别，它按优先级决定 `DirectInput` 将鼠标输入传递给程序的程度。要设置协作级别，可调用 `SetCooperativeLevel` 函数：

```
HRESULT SetCooperativeLevel(  
    HWND hwnd,  
    DWORD dwFlags  
);
```

第一个参数是窗口句柄。第二个是个有趣的参数，因为它指定程序对鼠标拥有的优先级。在鼠标上最常用的值是 `DISCL_NONEXCLUSIVE` 和 `DISCL_FOREGROUND`。调用这个函数的方法是：

```
HRESULT result = dimouse->SetCooperativeLevel(hwnd,  
    DISCL_NONEXCLUSIVE | DISCL_FOREGROUND);
```

## 3. 获取设备

最后一步是使用 `Acquire` 函数获取鼠标设备。如果函数返回 `DI_OK`，那么就说明成功地获取了鼠标，此时可以开始检查鼠标移动和按键了。

和键盘设备一样，在使用完鼠标后也得反获取鼠标设备，否则 `DirectInput` 会处于不稳定状态：

```
HRESULT Unacquire(VOID);
```

### 5.2.2 读取鼠标

需要在游戏循环的某个地方轮询鼠标以便更新鼠标位置及按钮状态。使用 `GetDeviceState` 函数来轮询鼠标：

```
HRESULT GetDeviceState(  
    DWORD cbData,  
    LPVOID lpvData  
);
```

第一个参数是用于填充数据的设备状态缓冲区的大小。第二个参数是指向该数据的指针。在轮询鼠标时可以使用这么一个结构：

```
DIMOUSESTATE mouse_state;
```

以下是调用 `GetDeviceState` 函数填充 `DIMOUSESTATE` 结构的方法：

```
dimouse->GetDeviceState(sizeof(mouse_state), (LPVOID)&mouse_state);
```

这个结构如下所示：

```
typedef struct DIMOUSESTATE {  
    LONG lX;  
    LONG lY;  
    LONG lZ;  
    BYTE rgbButtons[4];  
} DIMOUSESTATE;
```

如果想支持超过 4 个按钮的复杂鼠标设备，则还有另外一个结构可用。这时，按钮数组的尺寸增加一倍，但结构还是一样：

```
typedef struct DIMOUSESTATE2 {  
    LONG lX;  
    LONG lY;  
    LONG lZ;  
    BYTE rgbButtons[8];  
} DIMOUSESTATE2;
```

在轮询了鼠标之后，可以检查 `mouse_state` 结构，确定 `x` 和 `y` 的运动及按钮状态。可以使用 `lX` 和 `lY` 成员变量来检查鼠标移动（鼠标移动也称为 `mickey`）。`Mickey` 是什么？`Mickey` 表示鼠标运动，而不是绝对位置，所以如果想使用鼠标定位值来绘制自己的鼠标指针则必须保留老的位置值。`Mickey` 是处理鼠标运动的方便方法，因为用户在一个方向连续移动时鼠标可持续报告移动状态，即使“指针”到达屏幕的边缘。

如同结构中的内容所示，`rgbButtons` 数组保存按钮按下的结果。如果想检查某个特定按钮（从 0 开始，对应按钮 1），可以如下编写代码：

```
button_1 = obj.rgbButtons[0] & 0x80;
```

使用 `define` 是探测按钮状态的更为方便的方法：

```
#define BUTTON_DOWN(obj, button) (obj.rgbButtons[button] & 0x80)
```

使用 `define`，可以按如下方法检查按钮：

```
button_1 = BUTTON_DOWN(mouse_state, 0);
```

### 5.3 Xbox 360 控制器输入

大多数 Windows 用户没有在其 PC 上安装 Xbox 360 控制器，这是我们必须接受的事实。编写需要控制器才能控制的游戏会是个大错误，因为这将会把大多数的潜在受众拒之门外。愿意为你的游戏赴汤蹈火的游戏者（不论是一般的还是铁杆的）会很罕见。首先，他们要么需要购买一个带线的 Xbox 360 控制器，要么为 PC 购买无线适配器以便使用更常见的无线控制器。这两种控制器都能良好工作，但对于用户已有的键盘和鼠标来说，它会是额外的负担。而用户已经习惯于使用键盘和鼠标来操作 PC 游戏了——如果强迫要求使用控制器，那么对用户来说这就不仅仅是一点点的不适了。由于这个原因，我在任何游戏中都不仅仅对控制器提供支持。不过，不提供鼠标和键盘以外的对 Xbox 360 控制器（见图 5-1）的支持，也是不合理的！而这正是我们将在本章和所有后面的章节中所要做的。一旦开始使用熟悉的控制器（如果你是个控制台游戏者，谁不

是呢？），就很难回到使用无聊的老键盘和鼠标的日子了。



图 5-1 Xbox 360 控制器

可使用 XInput 库来获得对 Xbox 360 控制器的访问。这个库由 XInput.h 头文件和 XInput.lib 库文件组成。可使用如下语句在项目中加入头文件：

```
#include <xinput.h>
```

然后在游戏项目中用这种方法引用库文件：

```
#pragma comment(lib, "xinput.lib")
```

**建议** XInput 库应该已经自动与 DirectX SDK 一起安装了，所以无需做其他事情来获得对 XInput 库的访问。如果出现任何问题，则可以参阅附录 A，了解配置 Visual C++ 和 DirectX 的方法。

### 5.3.1 初始化 XInput

如果不在 PC 上接入控制器，那么在运行使用控制器的程序时，不用担心会发生问题，不会有错误信息出现。如果有控制器可用，那就可以开始读取其输入数据了。

以下是初始化 XInput 库的一种方法：获取控制器的能力并且检查这些值，确定控制器是否接入（不论是无线适配器还是接入 USB 口的有线控制器）：

```
XINPUT_CAPABILITIES caps;  
ZeroMemory(&caps, sizeof(XINPUT_CAPABILITIES));  
XInputGetCapabilities(0, XINPUT_FLAG_GAMEPAD, &caps);  
if (caps.Type != 0) return false;
```

如果在 PC 上接入了任何其他 Xbox 360 附件（比如游戏垫），那么这段只检查控制器的代码将报告失败信息。如果不想对这些不寻常的设备提供支持，则需要从 xinput.h 中找到该设备的类

型值。

**建议** 虽然 DirectInput 支持其他厂商（比如罗技）生产的游戏棒、游戏垫、方向盘及其他附件，但 XInput 库只支持 Xbox 360 附件，目前包括 Xbox 360 控制器、方向盘、摇杆、飞行杆、跳舞毯、吉他，甚至还有套鼓！

### 5.3.2 读取控制器状态

使用 XInput 的控制器状态结构和函数来读取 Xbox 360 控制器。这个结构名称为 XINPUT\_STATE，支持它的函数名为 XInputGetState。必须首先创建一个新的结构变量并将其清零：

```
XINPUT_STATE state;
ZeroMemory( &state, sizeof(XINPUT_STATE) );
```

然后通过调用 XInputGetState 函数读取控制器状态：

```
DWORD result = XInputGetState( 0, &state );
```

如果结果是零，就表示找到了控制器并且 XINPUT\_STATE 结构填入了数据。否则，如果结果不是零，则表示没找到控制器。

检查了 XInputGetState 的结果之后，可以使用如下代码查看 XINPUT\_STATE 变量中的属性：

```
if (state.Gamepad.bLeftTrigger)
    MessageBox(0, "Left Trigger", "Controller", 0);
```

XINPUT\_STATE 结构包含这些属性：

- DWORD dwPacketNumber;
- XINPUT\_GAMEPAD Gamepad;

它进一步分解到 XINPUT\_GAMEPAD 结构中，真正重要的属性都在里面：

- WORD wButtons;
- BYTE bLeftTrigger;
- BYTE bRightTrigger;
- SHORT sThumbLX;
- SHORT sThumbLY;
- SHORT sThumbRX;
- SHORT sThumbRY;

模拟触发器按钮（bLeftTrigger 和 bRightTrigger）生成 8 位的范围在 0 ~ 255 之间的数字。

模拟的“拇指”杆为每个轴（x 和 y）生成 16 位的范围在 -32 768（最左边）到 +32 767（最右边）的数字。每个杆中间的“死区”通常在  $\pm 1\,500$  个单元左右。所以，如果只想使用模拟杆实现方向按钮功能，最好是将其和  $\pm 5\,000$  左右的值进行比较（例如，if (sThumbLX < -5000) ...）。



wButtons 属性是个包含位掩码的数字，它组合了所有的按钮。

- XINPUT\_GAMEPAD\_DPAD\_UP
- XINPUT\_GAMEPAD\_DPAD\_DOWN
- XINPUT\_GAMEPAD\_DPAD\_LEFT
- XINPUT\_GAMEPAD\_DPAD\_RIGHT
- XINPUT\_GAMEPAD\_START
- XINPUT\_GAMEPAD\_BACK
- XINPUT\_GAMEPAD\_LEFT\_THUMB
- XINPUT\_GAMEPAD\_RIGHT\_THUMB
- XINPUT\_GAMEPAD\_LEFT\_SHOULDER
- XINPUT\_GAMEPAD\_RIGHT\_SHOULDER
- XINPUT\_GAMEPAD\_A
- XINPUT\_GAMEPAD\_B
- XINPUT\_GAMEPAD\_X
- XINPUT\_GAMEPAD\_Y

如果想检查某个按钮，则必须使用位与运算（&）将 wButtons 与定义的按钮值比较，以判断特定按钮是否按下。比如：

```
if (state.wButtons & XINPUT_GAMEPAD_DPAD_LEFT) ...
```

对于组合动作（比如 A+B 执行特殊攻击或特殊动作）的检查可通过包含多个按钮值来进行，但通常单独处理各个按钮按键或者使用 if 语句嵌套会更容易。

### 5.3.3 控制器振动

实际上，甚至可以通过一个非常简单的函数告诉控制器做振动动作！（也称为“力反馈”），可以使用 XINPUT\_VIBRATION 结构实现振动，它有两个属性：wLeftMotorSpeed 和 wRightMotorSpeed，其值可设为 0 到 65 535 之间（0 表示关闭，65 535 表示全振动，小心，这样做会让控制器从桌子上振下来！）。

```
XINPUT_VIBRATION vibration;
ZeroMemory( &vibration, sizeof(XINPUT_VIBRATION) );
vibration.wLeftMotorSpeed = left;
vibration.wRightMotorSpeed = right;
XInputSetState( 0, &vibration );
```

### 5.3.4 测试 XInput

这里有一个很简短的示例程序要和读者分享，这个程序名为 XInput Test。这个程序包含三个有帮助的函数，读者可在自己的 DirectX 游戏中通过它们来支持 Xbox 360 控制器。实际上，没有 DirectX 也可使用 XInput，因为这是个独立的库，根本不需要 DirectX 对象。唯一需要和 DirectX 链接在一起的情况是在使用接到控制器上的语音聊天耳机时（这时候 XInput 和 DirectSound 一起

工作)。所以可以为一个工作中的数据库应用程序添加控制器输入支持（建议使用按钮来执行数据库查询）。图 5-2 显示了简单的 XInput Test 程序，它只显示一个消息框，显示被按下的按钮的名称。

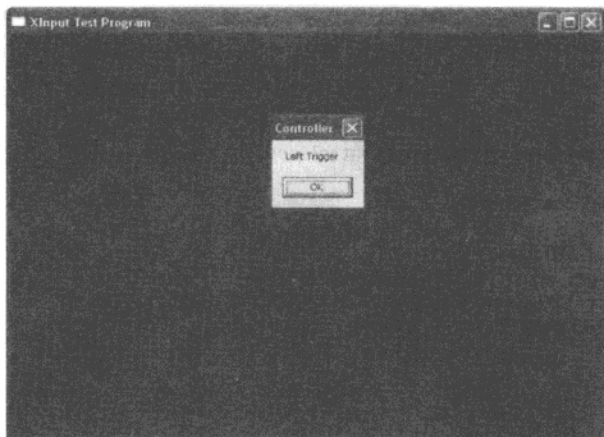


图 5-2 XInput Test 程序通过 MessageBox 报告控制器输入事件

XInput Test 程序会是最后一个完全列出完整源代码的程序。由于每个程序中都有大量重复的 Windows 代码，所以我们将把这个程序分解成许多小部分，以便后面马上要介绍的 Bomb Catcher 游戏使用。重要的新代码以粗体突出显示。

**建议** XInput Test 程序使用按钮 A 启动振动、按钮 B 停止振动、Back 按钮结束程序。使用 Esc 键仍旧可以退出。这会是我们将来所有程序的做法。

```
/*
    Beginning Game Programming, Third Edition
    Chapter 5
    XInput_Test Program
*/

#include <windows.h>
#include <d3d9.h>
#include <d3dx9.h>
#include <xinput.h>
#include <iostream>
using namespace std;

#pragma comment(lib, "d3d9.lib")
#pragma comment(lib, "d3dx9.lib")
#pragma comment(lib, "xinput.lib")

//application title
const string APPTITLE = "XInput Test Program";

//macro to read the keyboard
#define KEY_DOWN(vk_code) ((GetAsyncKeyState(vk_code) & 0x8000) ? 1 : 0)
```

```

//screen resolution
const int SCREENW = 640;
const int SCREENH = 480;

//Direct3D objects
LPDIRECT3D9 d3d = NULL;
LPDIRECT3DDEVICE9 d3ddev = NULL;

bool gameover = false;

/**
** Initializes XInput and any connected controllers
**/
bool XInput_Init(int contNum = 0)
{
    XINPUT_CAPABILITIES caps;
    ZeroMemory(&caps, sizeof(XINPUT_CAPABILITIES));
    XInputGetCapabilities(contNum, XINPUT_FLAG_GAMEPAD, &caps);
    if (caps.Type != XINPUT_DEVTYPE_GAMEPAD) return false;

    return true;
}

/**
** Causes the controller to vibrate
**/
void XInput_Vibrate(int contNum = 0, int left = 65535, int right = 65535)
{
    XINPUT_VIBRATION vibration;
    ZeroMemory(&vibration, sizeof(XINPUT_VIBRATION));
    vibration.wLeftMotorSpeed = left;
    vibration.wRightMotorSpeed = right;
    XInputSetState(contNum, &vibration);
}

/**
** Checks the state of the controller
**/
void XInput_Update()
{
    for (int i=0; i<4; i++)
    {
        XINPUT_STATE state;
        ZeroMemory(&state, sizeof(XINPUT_STATE));

        //get the state of the controller
        DWORD result = XInputGetState(i, &state);

        if( result == 0 )
        {
            // controller is connected
            if (state.Gamepad.bLeftTrigger)
                MessageBox(0, "Left Trigger", "Controller", 0);

            else if (state.Gamepad.bRightTrigger)
                MessageBox(0, "Right Trigger", "Controller", 0);
        }
    }
}

```

PDF

```
else if (state.Gamepad.sThumbLX < -10000 ||
state.Gamepad.sThumbLX > 10000)
    MessageBox(0, "Left Thumb Stick", "Controller", 0);

else if (state.Gamepad.sThumbRX < -10000 ||
state.Gamepad.sThumbRX > 10000)
    MessageBox(0, "Right Thumb Stick", "Controller", 0);

else if (state.Gamepad.wButtons & XINPUT_GAMEPAD_DPAD_UP)
    MessageBox(0, "DPAD Up", "Controller", 0);
else if (state.Gamepad.wButtons & XINPUT_GAMEPAD_DPAD_DOWN)
    MessageBox(0, "DPAD Down", "Controller", 0);

else if (state.Gamepad.wButtons & XINPUT_GAMEPAD_DPAD_LEFT)
    MessageBox(0, "DPAD Left", "Controller", 0);

else if (state.Gamepad.wButtons & XINPUT_GAMEPAD_DPAD_RIGHT)
    MessageBox(0, "DPAD Right", "Controller", 0);

else if (state.Gamepad.wButtons & XINPUT_GAMEPAD_START)
    MessageBox(0, "Start", "Controller", 0);

else if (state.Gamepad.wButtons & XINPUT_GAMEPAD_LEFT_THUMB)
    MessageBox(0, "Left Thumb", "Controller", 0);

else if (state.Gamepad.wButtons & XINPUT_GAMEPAD_RIGHT_THUMB)
    MessageBox(0, "Right Thumb", "Controller", 0);

else if (state.Gamepad.wButtons & XINPUT_GAMEPAD_LEFT_SHOULDER)
    MessageBox(0, "Left Shoulder", "Controller", 0);

else if (state.Gamepad.wButtons & XINPUT_GAMEPAD_RIGHT_SHOULDER)
    MessageBox(0, "Right Shoulder", "Controller", 0);

else if (state.Gamepad.wButtons & XINPUT_GAMEPAD_A)
{
    XInput_Vibrate(0, 65535, 65535);
    MessageBox(0, "A", "Controller", 0);
}

else if (state.Gamepad.wButtons & XINPUT_GAMEPAD_B)
{
    XInput_Vibrate(0, 0, 0);
    MessageBox(0, "B", "Controller", 0);
}

else if (state.Gamepad.wButtons & XINPUT_GAMEPAD_X)
    MessageBox(0, "X", "Controller", 0);

else if (state.Gamepad.wButtons & XINPUT_GAMEPAD_Y)
    MessageBox(0, "Y", "Controller", 0);

else if (state.Gamepad.wButtons & XINPUT_GAMEPAD_BACK)
    gameover = true;
```

```
    }
    else {
        // controller is not connected
    }
}

/**
** Game initialization
**/
bool Game_Init(HWND hwnd)
{
    //initialize Direct3D
    d3d = Direct3DCreate9(D3D_SDK_VERSION);
    if (d3d == NULL)
    {
        MessageBox(hwnd, "Error initializing Direct3D", "Error", MB_OK);
        return false;
    }

    //set Direct3D presentation parameters
    D3DPRESENT_PARAMETERS d3dpp;
    ZeroMemory(&d3dpp, sizeof(d3dpp));
    d3dpp.Windowed = TRUE;
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
    d3dpp.BackBufferFormat = D3DFMT_X8R8G8B8;
    d3dpp.BackBufferCount = 1;
    d3dpp.BackBufferWidth = SCREENW;
    d3dpp.BackBufferHeight = SCREENH;
    d3dpp.hDeviceWindow = hwnd;

    //create Direct3D device
    d3d->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hwnd,
        D3DCREATE_SOFTWARE_VERTEXPROCESSING, &d3dpp, &d3ddev);

    if (!d3ddev)
    {
        MessageBox(hwnd, "Error creating Direct3D device", "Error", MB_OK);
        return false;
    }

    //initialize XInput
    XInput_Init();
    return true;
}

/**
** Game update
**/
void Game_Run(HWND hwnd)
{
    //make sure the Direct3D device is valid
    if (!d3ddev) return;

    //clear the backbuffer to black
```

```

d3ddev->Clear(0, NULL, D3DCLEAR_TARGET, D3DCOLOR_XRGB(0,0,150), 1.0f, 0);

//start rendering
if (d3ddev->BeginScene())
{
    d3ddev->EndScene();
    d3ddev->Present(NULL, NULL, NULL, NULL);
}

//check for escape key (to exit program)
if (KEY_DOWN(VK_ESCAPE))
    PostMessage(hwnd, WM_DESTROY, 0, 0);

//check for controller input
XInput_Update();
}

/**
** Game shutdown
**/
void Game_End(HWND hwnd)
{
    if (d3ddev) d3ddev->Release();
    if (d3d) d3d->Release();
}

/**
** Windows event handler
**/
LRESULT WINAPI WinProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam )
{
    switch( msg )
    {
        case WM_DESTROY:
            gameover = true;
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc( hWnd, msg, wParam, lParam );
}

/**
** Windows entry point
**/
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{
    //create the window class structure
    WNDCLASSEX wc;
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc = (WNDPROC)WinProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;

```

```

wc.hInstance      = hInstance;
wc.hIcon          = NULL;
wc.hCursor        = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground  = (HBRUSH)GetStockObject(WHITE_BRUSH);
wc.lpszMenuName    = NULL;
wc.lpszClassName  = APPTITLE.c_str();
wc.hIconSm        = NULL;
RegisterClassEx(&wc);

//create a new window
HWND window = CreateWindow(APPTITLE.c_str(), APPTITLE.c_str(),
    WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
    SCREENW, SCREENH, NULL, NULL, hInstance, NULL);

//was there an error creating the window?
if (window == 0) return 0;

//display the window
ShowWindow(window, nCmdShow);
UpdateWindow(window);

//initialize the game
if (!Game_Init(window)) return 0;

// main message loop
MSG message;
while (!gameover)
{
    if (PeekMessage(&message, NULL, 0, 0, PM_REMOVE))
    {
        TranslateMessage(&message);
        DispatchMessage(&message);
    }
    Game_Run(window);
}
Game_End();
return message.wParam;
}

```

## 5.4 精灵编程简介

使用 Direct3D 绘制精灵有两种方法。这两种方法都要求我们保存精灵的位置、尺寸和速度，还要保存我们自己所需的其他属性，所以其逻辑是独立的。两种方法中简单的那种是将精灵图像装载到 D3D 表面中（在第 4 章已经学过），然后使用 StretchRect 绘制精灵。难一点但是更强大的方法是使用特殊的称为 D3DXSprite 的对象来处理 Direct3D 中的精灵。D3DXSprite 使用纹理而不是表面来保存精灵图像，所以如果使用它，那么需要使用与第 4 章稍微不一样的方法。不过，将位图图像装载到纹理中不比将图像装载到表面中难。本章将讲解简单的绘制精灵的方法，第 6 章讲解 D3DXSprite。

精灵是一个小的绘制在屏幕上的位图图像，代表游戏中的一个角色或对象。精灵可用在像树

木和岩石这样的静止对象上，也可以是诸如角色扮演游戏中的主人公这样的动画游戏角色。在现代游戏编程中有一件事情是肯定的：精灵只属于 2D 王国。在 3D 游戏中没有精灵，除非这个精灵被绘制在 3D 渲染的游戏场景“之上”，作为题头显示或位图字体。例如，在一个带有聊天功能的多人游戏中，来自其他游戏者的文本消息通常以单独的字母显示在屏幕上，每个字母都作为精灵对待。图 5-3 显示了储存于一个位图文件中的位图字体的示例。

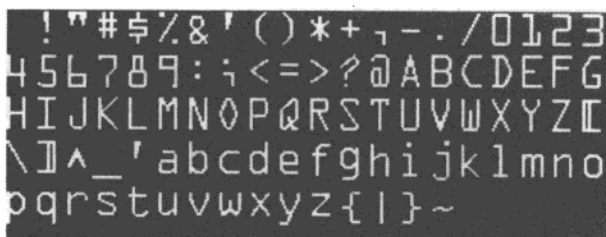


图 5-3 在游戏中用于在屏幕上打印文本的位图字体

精灵通常将一系列图片单元 (tile) 储存于一个位图文件中，每个图片单元表示该精灵动画序列中的一个帧。动画的效果更像方向的改变，而不是移动，如射击游戏中的飞机或飞船那样。图 5-4 显示了一个朝向单个方向但包含了动画履带来表示移动的坦克精灵。

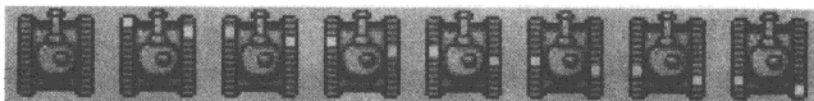


图 5-4 带有动画履带的坦克精灵。感谢 Ari Feldman 提供 ([www.flyingyogi.com](http://www.flyingyogi.com))

如果不仅想让坦克动，还要其朝向其他方向该怎么办？不难想象，如果为每个行进方向添加新的动画帧，那么帧的数量会以指数方式增加。图 5-5 显示了一个可以以非常平滑的旋转速率以 32

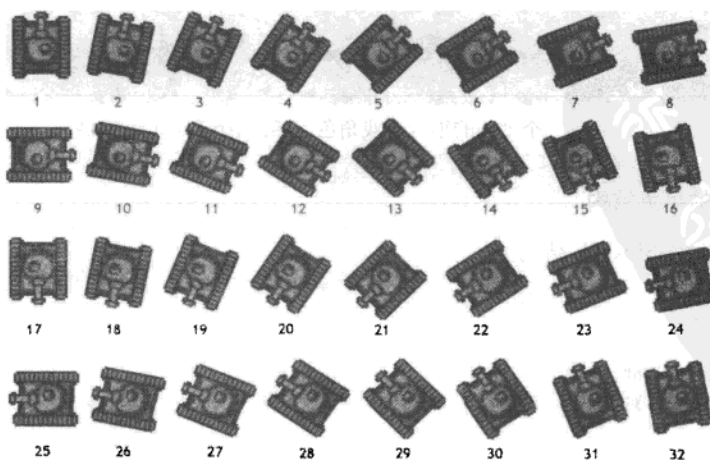


图 5-5 坦克精灵的 32 帧旋转（无动画），感谢 Ari Feldman 提供



个方向旋转的不动的坦克。遗憾的是，在添加移动的坦克履带时，这 32 帧一时间就会变成  $32 \times 8 = 256$  帧！给带有这么多帧的坦克编程会很困难，而且在位图文件中存储它们也是个问题。以行和列线性地存储是最有可能的方法。更好的解决方案通常是减少帧的数量，先完成游戏，然后再让动画更为精细。不过，所有这些只是推测，在第 6 章我们将学习如何使用代码实现精灵旋转！

MechCommander (MicroProse, FASA Studios) 是史上动画效果最强的视频游戏之一，如果不是因为游戏中糟糕的人工智能和不切实际的难关，我会认为它是我最喜欢的游戏之一。MechCommander 迷人的地方在于其高度精细的基于精灵的 2D 游戏。游戏中的每个机甲都是储存于一系列位图文件中的 2D 精灵。由于这个游戏有大约 100 000 个帧，其传统的 2D 本质变得令人惊奇！想象一下，先要使用 3D 建模软件（比如 3ds max）给机甲建模，然后将不同角度和位置的 100 000 个快照渲染出来，然后给每个精灵调整尺寸并且最后润色，这得花费多少时间啊！

**注意** 2006 年 8 月，Microsoft 发布了 MechCommander 2 的源代码及所有该游戏的资源（原图等）。游戏（由 DirectX 支持）的完整代码可以从这里下载：<http://www.microsoft.com/downloads/details.aspx?familyid=6D790CDE-C3E5-46BE-B3A5-729581269A9C&displaylang=en>。可以通过搜索“mechcommander 2 source code”找到这个链接。

另外一种常见的精灵类型是平台游戏精灵，如图 5-6 所示。编写平台游戏（platform game）要比射击游戏难，但终究物有所值。

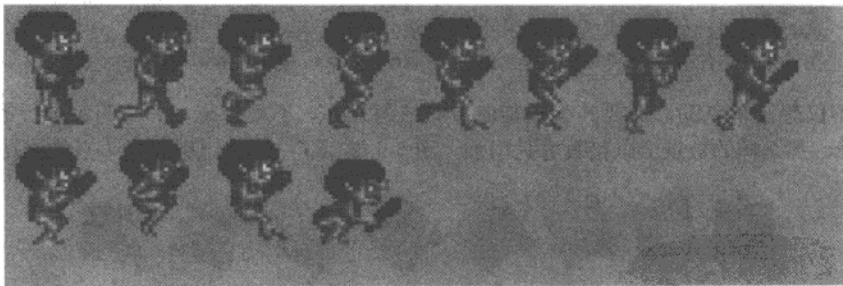


图 5-6 一个动画的平台游戏角色，感谢 Ari Feldman 提供

#### 5.4.1 一个有用的精灵结构

本程序的关键是 SPRITE 结构：

```
//sprite structure
struct SPRITE {
    float x,y;
    int width,height;
    float velx,vely;
}
```

这个结构显而易见的成员是 x、y、width 和 height。不那么显而易见的是 velx 和 vely。这些

成员变量用于在更新每帧时更新精灵的 x 和 y 位置。

以下这个示例展示了如何使用这个新结构创建一个精灵：

```
SPRITE spaceship;
spaceship.x = 100;
spaceship.y = 150;
spaceship.width = 96;
spaceship.height = 96;
spaceship.velx = 8.0f;
spaceship.vely = 0.0f;
```

#### 5.4.2 装载精灵图像

制作简单的精灵所需的一切就是这个结构和一个给图像准备的 Direct3D 表面。先复习一下表面装载和绘制代码。首先，再次看一下如何装载位图文件并将其储存到内存中的表面上（这些内容在第 4 章中讲过）：

```
//get width and height from bitmap file
D3DXIMAGE_INFO info;
HRESULT result = D3DXGetImageInfoFromFile(filename, &info);

//create surface
LPDIRECT3DSURFACE9 image = NULL;
result = d3ddev->CreateOffscreenPlainSurface(
    info.Width,           //width of the surface
    info.Height,          //height of the surface
    D3DFMT_X8R8G8B8,      //surface format
    D3DPPOOL_DEFAULT,     //memory pool to use
    &image,                //pointer to the surface
    NULL);                //reserved (always NULL)

//load surface from file into newly created surface
result = D3DXLoadSurfaceFromFile(
    image,                //destination surface
    NULL,                 //destination palette
    NULL,                 //destination rectangle
    filename,             //source filename
    NULL,                 //source rectangle
    D3DX_DEFAULT,         //controls how image is filtered
    transcolor,           //for transparency (0 for none)
    NULL);                //source image info (usually NULL)
```

#### 5.4.3 绘制精灵图像

在成功地将位图文件装载到内存中的 Direct3D 表面后，就可使用 StretchRect 函数来绘制图像了。假设已经创建了一个指向后台缓冲区的指针，那么绘制图像的方法就是：

```
//draw surface to the backbuffer
d3ddev->StretchRect(image, NULL, backbuffer, NULL, D3DTEXF_NONE);
```

这里的两个 NULL 参数是源和目标矩形，用于确切地指定图像来源位置并且确切地指定要绘制的目标位置。通常应该传递实际的矩形，以便精灵按照我们想要的方式正确显示出来。以下就是一个实现这一功能的有帮助的函数。这个函数使用 D3DSURFACE\_DESC 和 GetDesc() 函数来获取源位图的宽度和高度，以便将其正确地绘制在目标表面上。

```
void DrawSurface( LPDIRECT3DSURFACE9 dest, float x, float y,
    LPDIRECT3DSURFACE9 source)
{
    //get width/height from source surface
    D3DSURFACE_DESC desc;
    source->GetDesc(&desc);

    //create rects for drawing
    RECT source_rect = {0, 0, (long)desc.Width, (long)desc.Height };
    RECT dest_rect = { (long)x, (long)y, (long)x+desc.Width,
        (long)y+desc.Height};

    //draw the source surface onto the dest
    d3ddev->StretchRect(source, &source_rect, dest, &dest_rect,
        D3DTEXF_NONE);
}
```

**建议** 后面两章将使用基于纹理的精灵来替换这里的基于表面的精灵代码，它们有更好的性能，例如旋转和缩放！

## 5.5 Bomb Catcher 游戏

对输入设备进行编程所需的代码差不多就是这些了。准备好用这些知识来实践我们的第一个 DirectX 游戏了吗？我们将创建一个称为 Bomb Catcher 的伪游戏来演示到目前为止我们所学的一切。这个游戏要求用户在屏幕底部移动一个篮筐，抓住从顶部随机落下的炸弹。看看你能抓住多少，不能让炸弹碰到屏幕底部，小心！我们来盘点一下到目前为止所积累的技能：

- 初始化 Direct3D
- 将位图文件装载到内存
- 绘制位图
- 键盘输入
- 鼠标输入
- Xbox 360 控制器输入

在这么短的时间内有这么一个技能清单那是相当了不起了，我们还只是在第 5 章呢！但这些都是制作最简单的游戏所需的最小要求（还没有任何音响效果）。而且，必须承认的是，我们并不绝对需要支持控制器输入，但由于这是件非常容易做的事情，所以既然提到了，就把它做出来吧。

图 5-7 显示了进行中的 Bomb Catcher 游戏。这个游戏支持键盘、鼠标及控制器输入，而且可以由读者自己来改进！更重要的是，这是第一个将 Windows、DirectX 和游戏源代码分开存放在

不同文件中的程序：

MyWindows.cpp  
MyDirectX.h  
MyDirectX.cpp  
MyGame.cpp

所有 Windows 代码，包括 WinMain 和 WinProc  
DirectX 变量和函数定义  
DirectX 变量和函数实现  
游戏的源代码

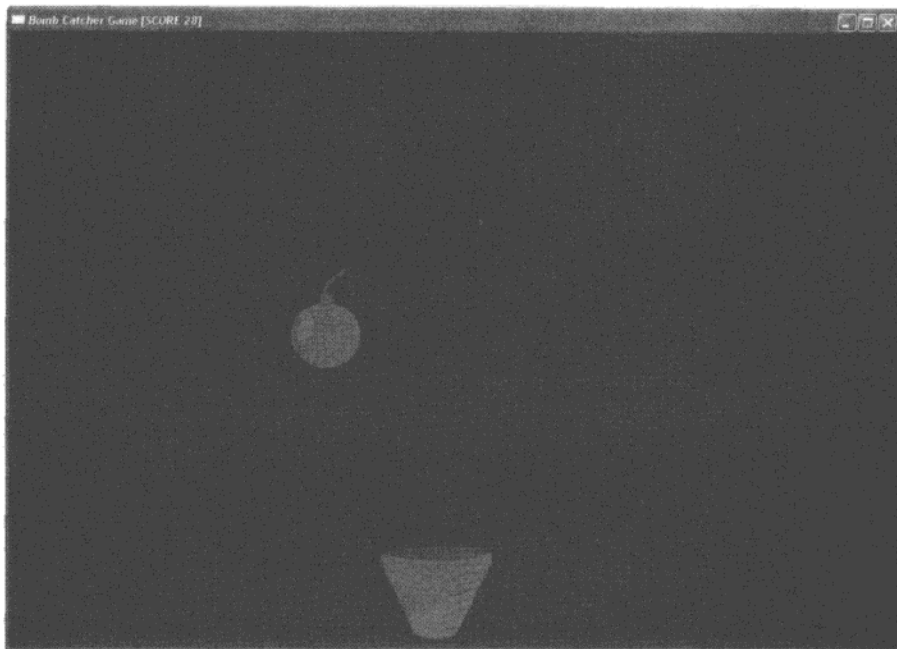


图 5-7 Bomb Catcher 游戏演示了鼠标、键盘和控制器输入

要成为专业程序员，代码重用是关键。不能一次一次重写代码并指望每一次都能把工作做好。到目前为止我们所创建的源代码文件提供了一个可极大减少编写 Windows/DirectX 游戏所需工作量的游戏框架。而且我们所说的是真正的 Direct3D 游戏！什么？我们甚至都还没进入 3D 呢？

我一直把 3D 压着不放是因为，它有点复杂。我想让大家把代码基础（框架）先准备好，然后再投入到 3D 代码中，因为要是不这样我们就会淹没在大量代码中。后面关于 3D 的章节（从第 12 章“学习 3D 渲染的基础”开始）将易于理解和掌握，因为本书不会让你陷入任何 3D 数学中，不过在使用 Direct3D 时还是会涉及大量的代码。

现在创建新的 Bomb Catcher 项目并将这些新源代码文件添加到项目中。我们将分别讲解这些源代码文件。

### 5.5.1 MyWindows.cpp

在新的 Win32 项目中添加一个新的源代码文件并将其命名为 MyWindows.cpp，如图 5-8 所示。这个源代码文件包含 WinProc 和 WinMain 函数中的所有标准 Windows 代码。由于这已经是

我们在本书中第 5 次给出这些清单了，所以这是最后一次给出了。保留这个源代码文件，因为我们不会再复制它了。

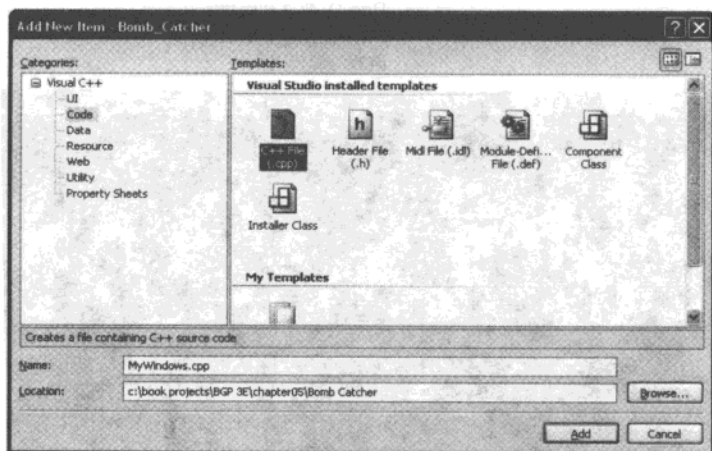


图 5-8 向项目中添加 MyWindows.cpp 文件

```

/*
    Beginning Game Programming, Third Edition
    MyWindows.cpp
*/

#include "MyDirectX.h"
using namespace std;
bool gameover = false;

/**
    ** Windows event handler
    **/
LRESULT WINAPI WinProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam )
{
    switch( msg )
    {
        case WM_DESTROY:
            gameover = true;
            PostQuitMessage(0);
            return 0;
    }
    return DefWindowProc( hWnd, msg, wParam, lParam );
}

/**
    ** Windows entry point
    **/
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
{

```

```
//initialize window settings
WNDCLASSEX wc;
wc.cbSize = sizeof(WNDCLASSEX);
wc.style = CS_HREDRAW | CS_VREDRAW;
wc.lpfnWndProc = (WNDPROC)WinProc;
wc.cbClsExtra = 0;
wc.cbWndExtra = 0;
wc.hInstance = hInstance;
wc.hIcon = NULL;
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
wc.lpszMenuName = NULL;
wc.lpszClassName = APPTITLE.c_str();
wc.hIconSm = NULL;
RegisterClassEx(&wc);

//create a new window
HWND window = CreateWindow(APPTITLE.c_str(), APPTITLE.c_str(),
    WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
    SCREENW, SCREENH, NULL, NULL, hInstance, NULL);
if (window == 0) return 0;

//display the window
ShowWindow(window, nCmdShow);
UpdateWindow(window);

//initialize the game
if (!Game_Init(window)) return 0;

// main message loop
MSG message;
while (!gameover)
{
    if (PeekMessage(&message, NULL, 0, 0, PM_REMOVE))
    {
        TranslateMessage(&message);
        DispatchMessage(&message);
    }

    //process game loop
    Game_Run(window);
}

//shutdown
Game_End();
return message.wParam;
}
```

### 5.5.2 MyDirectX.h

在新的 Win32 项目中添加一个新源代码文件并命名为 MyDirectX.h, 如图 5-9 所示。这个文件包含制作 DirectX 游戏所需的所有函数原型、头文件、库文件及全局变量! 我们会在添加新功

能时在这个文件中增加内容（比如第 6 章中的纹理装载）。

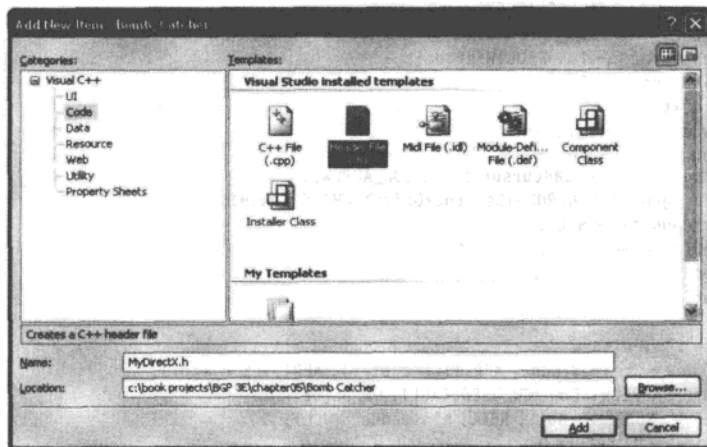


图 5-9 向项目中添加 MyDirectX.h 文件

```

/*
    Beginning Game Programming, Third Edition
    MyDirectX.h
*/

#pragma once

//header files
#define WIN32_EXTRA_LEAN
#define DIRECTINPUT_VERSION 0x0800
#include <windows.h>
#include <d3d9.h>
#include <d3dx9.h>
#include <dinput.h>
#include <xinput.h>
#include <ctime>
#include <iostream>
#include <iomanip>
using namespace std;

//libraries
#pragma comment(lib, "winmm.lib")
#pragma comment(lib, "user32.lib")
#pragma comment(lib, "gdi32.lib")
#pragma comment(lib, "dxguid.lib")
#pragma comment(lib, "d3d9.lib")
#pragma comment(lib, "d3dx9.lib")
#pragma comment(lib, "dinput8.lib")
#pragma comment(lib, "xinput.lib")

//program values
extern const string APPTITLE;

```

```

extern const int SCREENW;
extern const int SCREENH;
extern bool gameover;

//Direct3D objects
extern LPDIRECT3D9 d3d;
extern LPDIRECT3DDEVICE9 d3ddev;
extern LPDIRECT3DSURFACE9 backbuffer;

//Direct3D functions
bool Direct3D_Init(HWND hwnd, int width, int height, bool fullscreen);
void Direct3D_Shutdown();
LPDIRECT3DSURFACE9 LoadSurface(string filename);
void DrawSurface(LPDIRECT3DSURFACE9 dest, float x, float y,
    LPDIRECT3DSURFACE9 source);

//DirectInput objects, devices, and states
extern LPDIRECTINPUT8 dinput;
extern LPDIRECTINPUTDEVICE8 dimouse;
extern LPDIRECTINPUTDEVICE8 dikeyboard;
extern DIMOUSESTATE mouse_state;
extern XINPUT_GAMEPAD controllers[4];

//DirectInput functions
bool DirectInput_Init(HWND);
void DirectInput_Update();
void DirectInput_Shutdown();
int Key_Down(int);
int Mouse_Button(int);
int Mouse_X();
int Mouse_Y();
void XInput_Vibrate(int contNum = 0, int amount = 65535);
bool XInput_Controller_Found();
//game functions
bool Game_Init(HWND window);
void Game_Run(HWND window);
void Game_End();

```

### 5.5.3 MyDirectX.cpp

在新的 Win32 项目中添加一个新源代码文件并命名为 MyDirectX.cpp, 如图 5-10 所示。这个文件包含游戏项目的 DirectX 实现。所有的 DirectX 代码都将包含在此, 包括 Direct3D 和 DirectInput。是的, 放在同一个源文件中可以保持简洁。在每个新的章节中, 随着我们在游戏编程工具集中增加新的技能, 我们将在本文件中添加新内容。

```

/*
    Beginning Game Programming, Third Edition
    MyDirectX.cpp
*/

#include "MyDirectX.h"
#include <iostream>
using namespace std;

```



```

//Direct3D variables
LPDIRECT3D9 d3d = NULL;
LPDIRECT3DDEVICE9 d3ddev = NULL;
LPDIRECT3DSURFACE9 backbuffer = NULL;

//DirectInput variables
LPDIRECTINPUT8 dinput = NULL;
LPDIRECTINPUTDEVICE8 dimouse = NULL;
LPDIRECTINPUTDEVICE8 dikeyboard = NULL;
DIMOUSESTATE mouse_state;
char keys[256];
XINPUT_GAMEPAD controllers[4];

/**
 ** Direct3D initialization
 **/
bool Direct3D_Init(HWND window, int width, int height, bool fullscreen)
{
    //initialize Direct3D
    d3d = Direct3DCreate9(D3D_SDK_VERSION);
    if (!d3d) return false;

    //set Direct3D presentation parameters
    D3DPRESENT_PARAMETERS d3dpp;
    ZeroMemory(&d3dpp, sizeof(d3dpp));
    d3dpp.Windowed = (!fullscreen);
    d3dpp.SwapEffect = D3DSWAPEFFECT_COPY;
    d3dpp.BackBufferFormat = D3DFMT_X8R8G8B8;
    d3dpp.BackBufferCount = 1;
    d3dpp.BackBufferWidth = width;
    d3dpp.BackBufferHeight = height;
    d3dpp.hDeviceWindow = window;

    //create Direct3D device
    d3d->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, window,
        D3DCREATE_SOFTWARE_VERTEXPROCESSING, &d3dpp, &d3ddev);
    if (!d3ddev) return false;

    //get a pointer to the back buffer surface
    d3ddev->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, &backbuffer);

    return true;
}

/**
 ** Direct3D shutdown
 **/
void Direct3D_Shutdown()
{
    if (d3ddev) d3ddev->Release();
    if (d3d) d3d->Release();
}

/**
 ** Draws a surface to the screen using StretchRect

```



```

        //make sure file was loaded okay
        if (result != D3D_OK) return NULL;

        return image;
    }

    /**
     ** DirectInput initialization
     **/
    bool DirectInput_Init(HWND hwnd)
    {
        //initialize DirectInput object
        HRESULT result = DirectInput8Create(
            GetModuleHandle(NULL),
            DIRECTINPUT_VERSION,
            IID_IDirectInput8,
            (void**)&diinput,
            NULL);

        //initialize the keyboard
        diinput->CreateDevice(GUID_SysKeyboard, &dikeyboard, NULL);
        dikeyboard->SetDataFormat(&c_dfDIKeyboard);
        dikeyboard->SetCooperativeLevel(hwnd, DISCL_NONEXCLUSIVE |
            DISCL_FOREGROUND);
        dikeyboard->Acquire();

        //initialize the mouse
        diinput->CreateDevice(GUID_SysMouse, &dimouse, NULL);
        dimouse->SetDataFormat(&c_dfDIMouse);
        dimouse->SetCooperativeLevel(hwnd, DISCL_NONEXCLUSIVE |
            DISCL_FOREGROUND);
        dimouse->Acquire();
        d3ddev->ShowCursor(false);

        return true;
    }

    /**
     ** DirectInput update
     **/
    void DirectInput_Update()
    {
        //update mouse
        dimouse->GetDeviceState(sizeof(mouse_state), (LPVOID)&mouse_state);

        //update keyboard
        dikeyboard->GetDeviceState(sizeof(keys), (LPVOID)&keys);

        //update controllers
        for (int i=0; i<4; i++)
        {
            ZeroMemory(&controllers[i], sizeof(XINPUT_STATE));

            //get the state of the controller

```

```
XINPUT_STATE state;
DWORD result = XInputGetState( i, &state );

//store state in global controllers array
if (result == 0) controllers[i] = state.Gamepad;
}

/**
** Return mouse x movement
**/
int Mouse_X()
{
    return mouse_state.lX;
}

/**
** Return mouse y movement
**/
int Mouse_Y()
{
    return mouse_state.lY;
}

/**
** Return mouse button state
**/
int Mouse_Button(int button)
{
    return mouse_state.rgbButtons[button] & 0x80;
}

/**
** Return key press state
**/
int Key_Down(int key)
{
    return (keys[key] & 0x80);
}

/**
** DirectInput shutdown
**/
void DirectInput_Shutdown()
{
    if (dikeyboard)
    {
        dikeyboard->Unacquire();
        dikeyboard->Release();
        dikeyboard = NULL;
    }
    if (dimouse)
    {

```

```

        dimouse->Unacquire();
        dimouse->Release();
        dimouse = NULL;
    }

}

/**
** Returns true if controller is plugged in
**/
bool XInput_Controller_Found()
{
    XINPUT_CAPABILITIES caps;
    ZeroMemory(&caps, sizeof(XINPUT_CAPABILITIES));
    XInputGetCapabilities(0, XINPUT_FLAG_GAMEPAD, &caps);
    if (caps.Type != 0) return false;

    return true;
}

/**
** Vibrates the controller
**/
void XInput_Vibrate(int contNum, int amount)
{
    XINPUT_VIBRATION vibration;
    ZeroMemory(&vibration, sizeof(XINPUT_VIBRATION));
    vibration.wLeftMotorSpeed = amount;
    vibration.wRightMotorSpeed = amount;
    XInputSetState(contNum, &vibration);
}

```

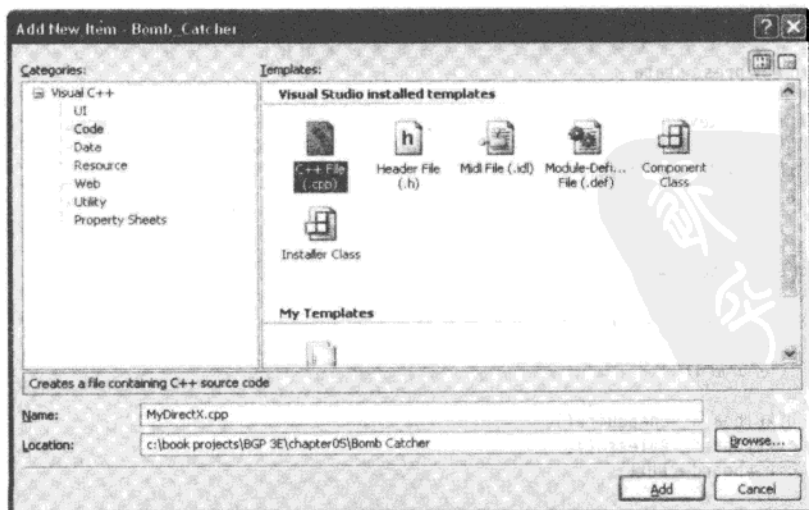


图 5-10 向项目中添加 MyDirectX.cpp 文件

### 5.5.4 MyGame.cpp

在新的 Win32 项目中添加一个新源代码文件并命名为 MyGame.cpp，如图 5-11 所示。这是 Bomb Catcher 游戏的主源代码文件，也是在将来的示例程序中需要编辑的文件。我们在添加新的对 DirectX 功能的支持时，这些支持代码主要进入 MyDirectX.h 和 MyDirectX.cpp 文件中；而这里的主文件（MyGame.cpp）将主要包含游戏逻辑，也就是重要的东西。虽然我们将程序分为许多文件，但这个文件还是很大的。这主要是由于对控制器输入过于热情的支持所致。在游戏中，可使用控制器上任何一对按钮来移动篮子（抓住炸弹）。对于游戏来讲这并不正常，但我们这么做是想在真实的游戏环境中展示使用这些按钮的方法。

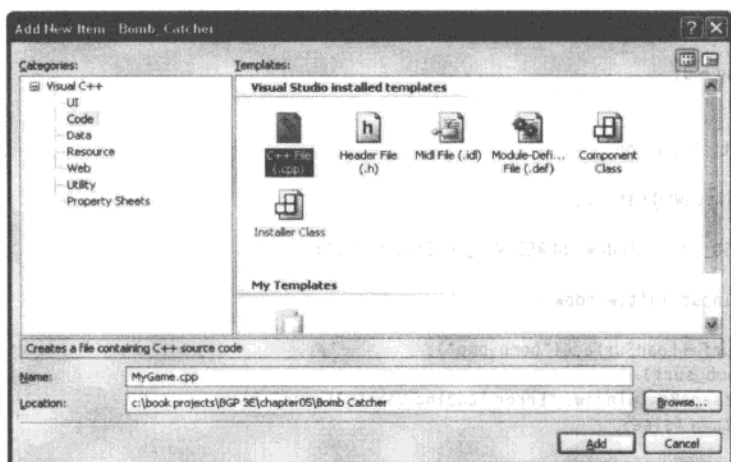


图 5-11 向项目中添加 MyGame.cpp 文件

```
/*
    Beginning Game Programming, Third Edition
    MyGame.cpp
*/

#include "MyDirectX.h"
using namespace std;

const string APPTITLE = "Bomb Catcher Game";
const int SCREENW = 1024;
const int SCREENH = 768;

LPDIRECT3DSURFACE9 bomb_surf = NULL;
LPDIRECT3DSURFACE9 bucket_surf = NULL;

struct BOMB
{
    float x,y;

    void reset()
```



```
{
    x = (float)(rand() % (SCREENW-128));
    y = 0;
}
};
BOMB bomb;

struct BUCKET
{
    float x,y;
};

BUCKET bucket;

int score = 0;
int vibrating = 0;

/**
 ** Game initialization
 **/
bool Game_Init(HWND window)
{
    Direct3D_Init(window, SCREENW, SCREENH, false);

    DirectInput_Init(window);

    bomb_surf = LoadSurface("bomb.bmp");
    if (!bomb_surf) {
        MessageBox(window, "Error loading bomb", "Error", 0);
        return false;
    }

    bucket_surf = LoadSurface("bucket.bmp");
    if (!bucket_surf) {
        MessageBox(window, "Error loading bucket", "Error", 0);
        return false;
    }

    //get the back buffer surface
    d3ddev->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, &backbuffer);

    //position the bomb
    srand( (unsigned int)time(NULL) );
    bomb.reset();

    //position the bucket
    bucket.x = 500;
    bucket.y = 630;
    return true;
}

/**
 ** Game update
 **/
```

```

** Game update
**/
void Game_Run(HWND window)
{
    //make sure the Direct3D device is valid
    if (!d3ddev) return;

    //update input devices
    DirectInput_Update();

    //move the bomb down the screen
    bomb.y += 2.0f;

    //see if bomb hit the floor
    if (bomb.y > SCREENH)
    {
        MessageBox(0, "Oh no, the bomb exploded!!", "YOU STINK", 0);
        gameover = true;
    }

    //move the bucket with the mouse
    int mx = Mouse_X();
    if (mx < 0) bucket.x -= 6.0f;
    else if (mx > 0) bucket.x += 6.0f;

    //move the bucket with the keyboard
    if (Key_Down(DIK_LEFT)) bucket.x -= 6.0f;
    else if (Key_Down(DIK_RIGHT)) bucket.x += 6.0f;

    //move the bucket with the controller
    if (XInput_Controller_Found())
    {
        //left analog thumb stick
        if (controllers[0].sThumbLX < -5000)
            bucket.x -= 6.0f;
        else if (controllers[0].sThumbLX > 5000)
            bucket.x += 6.0f;

        //left and right triggers
        if (controllers[0].bLeftTrigger > 128)
            bucket.x -= 6.0f;
        else if (controllers[0].bRightTrigger > 128)
            bucket.x += 6.0f;

        //left and right D-PAD
        if (controllers[0].wButtons & XINPUT_GAMEPAD_LEFT_SHOULDER)
            bucket.x -= 6.0f;
        else if (controllers[0].wButtons & XINPUT_GAMEPAD_RIGHT_SHOULDER)
            bucket.x += 6.0f;

        //left and right shoulders
        if (controllers[0].wButtons & XINPUT_GAMEPAD_DPAD_LEFT)
            bucket.x -= 6.0f;
        else if (controllers[0].wButtons & XINPUT_GAMEPAD_DPAD_RIGHT)

```



```
        bucket.x += 6.0f;
    }

    //update vibration
    if (vibrating > 0)
    {
        vibrating++;
        if (vibrating > 20)
        {
            XInput_Vibrate(0, 0);
            vibrating = 0;
        }
    }

    //keep bucket inside the screen
    if (bucket.x < 0) bucket.x = 0;
    if (bucket.x > SCREENW-128) bucket.x = SCREENW-128;

    //see if bucket caught the bomb
    int cx = bomb.x + 64;
    int cy = bomb.y + 64;
    if (cx > bucket.x && cx < bucket.x+128 &&
        cy > bucket.y && cy < bucket.y+128)
    {
        //update and display score
        score++;
        char s[255];
        sprintf(s, "%s [SCORE %d]", APPTITLE.c_str(), score);
        SetWindowText(window, s);

        //vibrate the controller
        XInput_Vibrate(0, 65000);
        vibrating = 1;

        //restart bomb
        bomb.reset();
    }

    //clear the backbuffer
    d3ddev->ColorFill(backbuffer, NULL, D3DCOLOR_XRGB(0,0,0));

    //start rendering
    if (d3ddev->BeginScene())
    {
        //draw the bomb
        DrawSurface(backbuffer, bomb.x, bomb.y, bomb_surf);

        //draw the bucket
        DrawSurface(backbuffer, bucket.x, bucket.y, bucket_surf);

        //stop rendering
        d3ddev->EndScene();
        d3ddev->Present(NULL, NULL, NULL, NULL);
    }
}
```

```
//escape key exits
if (Key_Down(DIK_ESCAPE))
    gameover = true;

//controller Back button also exits
if (controllers[0].wButtons & XINPUT_GAMEPAD_BACK)
    gameover = true;
}

/**
** Game shutdown
**/
void Game_End()
{
    if (bomb_surf) bomb_surf->Release();
    if (bucket_surf) bucket_surf->Release();
    DirectInput_Shutdown();
    Direct3D_Shutdown();
}
```

## 5.6 你所学到的

本章学习了如何使用 DirectInput 进行键盘、鼠标和控制器编程及如何在 Direct3D 中对 2D 表面和精灵进行编程，并且小有成就！如果你对所有这些新知识还没建立起自信，不要气馁，因为我们所学的可不是简单的技艺！如果有任何疑问，这里建议读者在进入高级精灵编程的第 6 章之前重读本章内容。不要对这里讨论的 2D 图形犹豫不决，要继续学习，因为这是即将到来的 3D 章节的基础！本章有以下几个要点：

- 如何初始化 DirectInput。
- 如何创建键盘处理器。
- 如何创建鼠标处理器。
- 如何对 Xbox 360 控制器进行编程。
- 编写一个名为 Bomb Catcher 的示例游戏。
- 关于精灵碰撞的知识。
- 如何创建由 Direct3D 渲染的 2D 表面。

## 5.7 复习测验

下面这些问题将考验你是不是还需要进一步学习本章知识。

- 1) 主 DirectInput 对象的名称是什么？
- 2) 创建 DirectInput 设备的函数是什么？
- 3) 包含鼠标输入数据的结构的名称是什么？
- 4) 轮询键盘或鼠标时调用的函数是哪个？
- 5) 帮助检查精灵碰撞的函数名称是什么？
- 6) 游戏的概念图能带来什么好处？



- 7) Direct3D 中的表面对象的名称是什么?
- 8) 将表面绘制在屏幕上应该用什么函数?
- 9) 将位图图像装载到表面中的 D3DX 助手函数是哪个?
- 10) 在 Web 上的哪儿能找到不错的免费精灵集?

## 5.8 自己动手

以下练习将帮助你跳出思维框框，最大限度地了解本章内容。

**习题 1** Bomb Catcher 项目还不是个完整的游戏，因为它只是输入演示而已，不过它很好地完成了自己作为一个简单演示程序的角色。正是这个简单演示的本质让它很有可能成为一个游戏项目，因为它无需从头开始做起。你是否能为其加入分数、生命和其他游戏功能，让它不那么像个演示，而是更像个游戏。你可能需要多阅读几章，学习一些新的技术，然后回到这里来改进这个游戏。

**习题 2** 你是否能增强 Bomb Catcher 尚待观察，但我们在短时间内能实现的是，在演示中添加另外一个炸弹，让游戏者抓住两个炸弹，而不只是一个！



## 第6章 绘制精灵并显示精灵动画

本章进一步讲解精灵编程这一主题。利用纹理技术而不是表面来处理精灵图像，就有可能绘制出有透明效果的精灵（只显示对象本身的像素而不显示背景。在 Bomb Catcher 中连背景都显示）。使用这种新的绘制精灵的方式，还能实现一些特殊效果，例如旋转和缩放。这些东西在计算机图形专家的嘴里称为“变换”，我们将用两章的篇幅来学习这些特殊效果。我们将开发一个健壮的、可重用的精灵函数集，在将来的游戏项目中使用。本章首先讲解基础知识，后面会讨论精灵动画。

本章将学到：

- 精灵是什么。
- 如何装载纹理。
- 如何绘制透明精灵。
- 如何使精灵动起来。

### 6.1 什么是精灵

精灵是游戏实体的 2D 表示，它通常必须以某种方式和游戏者交互。树木和岩石可以以 2D 方式渲染并且以挡道的方式和游戏者交互；以物理碰撞的方式停住游戏者。第 5 章我们使用 Direct3D 表面绘制精灵。而表面最大的问题是除了绘图速度很慢以外，还缺乏对任何透明类型的支持（这是我们的炸弹和篮子精灵有黑色轮廓的原因）。

我们还必须处理那些直接或间接与游戏者的角色交互的游戏角色（例如飞船、意大利水管工或者带刺的刺猬）。在太空战斗游戏中，这类与游戏者交互的精灵可能会是敌方的飞船或者激光炮，下面将陆续用示例来说明。

在 Direct3D 中有两种渲染 2D 对象的方法。第一种方法是创建一个由两个三角形组成的四边形（quad，也就是矩形），这两个三角形带有纹理，表示想要绘制的 2D 图形。这种技术不仅有效，而且甚至支持透明、响应光照并且可在 Z 方向上移动。第二种在 Direct3D 中可用于渲染 2D 对象的方法是使用精灵，这是我们将在本章重点讨论的方法。

### 6.2 装载精灵图像

首先必须知道的是，ID3DXSprite 使用纹理而不是表面来储存精灵图像。所以，虽然第 5 章我们使用 LPDIRECT3DSURFACE9 对象来制作精灵，但本章将使用 LPDIRECT3DTEXTURE9 对象。如果创建基于图片单元卷动的街机游戏，例如 Super Mario World 或者 R-Type 或者 Mars Matrix，就需要使用表面来绘制（并且卷动）背景，但会根据情况使用纹理来实现前景上代表游戏角色 / 飞船 / 敌人的精灵。使用表面而不使用纹理实在是对性能没有好处，因为昂贵的视频卡（带有高级 3D 芯片）将在屏幕上使用硬件纹理映射系统来渲染精灵，它要比使用软件快上好

几光年。那些使用汇编语言将 2D 精灵传输到屏幕上的日子已经一去不复返了！今天，我们让 Direct3D 来绘制精灵。这也正是我们将从现在开始使用纹理的原因。

创建游戏精灵首先要做的是创建一个用于装载精灵位图图像的纹理对象：

```
LPDIRECT3DTEXTURE9 texture = NULL;
```

而后需要做的是使用 D3DXGetImageInfoFromFile 函数从位图文件中取出分辨率数据（假设已经有可用的精灵位图）：

```
D3DXIMAGE_INFO info;
result = D3DXGetImageInfoFromFile("image.bmp", &info);
```

如果文件存在，就会获得图像的 Width（宽度）和 Height（高度），这对下一步很有用。尤其在开始一个游戏项目时，有了这些信息真是很方便。所以，我们可以编写一个可重用的函数，以 D3DXVECTOR2（它有包含图像宽度和高度信息的 X 和 Y 属性）返回一个图像的尺寸。如果想使用它，可将这个函数加入到 MyDirectX.h 和 MyDirectX.cpp 文件中。

```
D3DXVECTOR2 GetBitmapSize(string filename)
{
    D3DXIMAGE_INFO info;
    D3DXVECTOR2 size = D3DXVECTOR2(0.0f, 0.0f);

    HRESULT result = D3DXGetImageInfoFromFile(filename.c_str(), &info);
    if (result == D3D_OK)
        size = D3DXVECTOR2((float)info.Width, (float)info.Height);
    else
        size = D3DXVECTOR2((float)info.Width, (float)info.Height);

    return size;
}
```

接下来，使用 D3DXCreateTextureFromFileEx 函数一步直接从位图文件中将精灵的图像装载到纹理中：

```
HRESULT WINAPI D3DXCreateTextureFromFileEx(
    LPDIRECT3DDEVICE9 pDevice,
    LPCWSTR pSrcFile,
    UINT Width,
    UINT Height,
    UINT MipLevels,
    DWORD Usage,
    D3DFORMAT Format,
    D3DPPOOL Pool,
    DWORD Filter,
    DWORD MipFilter,
    D3DCOLOR ColorKey,
    D3DXIMAGE_INFO *pSrcInfo,
    PALETTEENTRY *pPalette,
    LPDIRECT3DTEXTURE9 *ppTexture
);
```



不用过多担心这么多的参数，它们当中的大多数都可用默认值和 NULL 来填写。唯一要做的事情，就是编写一个小函数，将所有这些信息放到一起并返回一个纹理。以下就是这样一个函数，将其命名为 LoadTexture（是不是很有创意？）：

```
LPDIRECT3DTEXTURE9 LoadTexture(string filename, D3DCOLOR transcolor)
{
    LPDIRECT3DTEXTURE9 texture = NULL;

    //get width and height from bitmap file
    D3DXIMAGE_INFO info;
    HRESULT result = D3DXGetImageInfoFromFile(filename.c_str(), &info);
    if (result != D3D_OK) return NULL;
    //create the new texture by loading a bitmap image file
    D3DXCreateTextureFromFileEx(
        d3ddev,           //Direct3D device object
        filename.c_str(), //bitmap filename
        info.Width,       //bitmap image width
        info.Height,      //bitmap image height
        1,                //mip-map levels (1 for no chain)
        D3DPPOOL_DEFAULT, //the type of surface (standard)
        D3DFMT_UNKNOWN,   //surface format (default)
        D3DPPOOL_DEFAULT, //memory class for the texture
        D3DX_DEFAULT,     //image filter
        D3DX_DEFAULT,     //mip filter
        transcolor,        //color key for transparency
        &info,              //bitmap file info (from loaded file)
        NULL,              //color palette
        &texture );        //destination texture

    //make sure the bitmap texture was loaded correctly
    if (result != D3D_OK) return NULL;

    return texture;
}
```

### 6.3 透明的精灵

ID3DXSprite 对象对于计划使用 Direct3D 编写 2D 游戏的程序员来说，实在是个极大的惊喜。它的好处就是，在使用和以前的实现（例如，老的 DirectDraw）一样快的 2D 函数的同时有完整的 3D 渲染器随时为你服务。将精灵处理成纹理并且以矩形（由两个三角形组成，如所有的 3D 矩形那样）来渲染，我们就可以对精灵进行变换！

关于变换，意思是指可以使用完整的 3D 硬件加速能力来移动精灵。通过在源位图中指定表示透明像素的 alpha 颜色我们可以绘制透明的精灵。黑色 (0,0,0) 是常见的用于透明的颜色，但它并不是非常理想的用色。为什么呢？因为用黑色就难以区分哪些像素是透明的，哪些像素只是颜色深而已。使用粉红色 (255,0,255) 是个更好的选择，因为它在游戏图形中很少用到，并且在源图像中显得明亮。这样我们可以立即看到透明像素。

显然，ID3DXSprite 方法是我们要用的方法，不过我也将讲解更简单的那种方法，因为在某些情况下使用非透明图像也会有帮助，比如，绘制使用图片单元平铺的背景。第 7 章我们将学习

如何使用 Direct3D 矩阵来实现精灵变换（也就是旋转、缩放和平移）。我们完全可以使用运行于视频卡上的 GPU（图形处理单元）中的顶点着色引擎来对精灵进行变换！高端的 GPU 能处理多个精灵，数都数不过来。

### 6.3.1 初始化精灵渲染器

ID3DXSprite 对象只是个包含从纹理绘制精灵的函数的精灵处理器（带有多种变换）。D3DX 定义了该类的指针版本：LPD3DXSPRITE，对我们来说它更为方便。以下是声明它的方法：

```
LPD3DXSPRITE spriteobj = NULL;
```

**建议** 在定义新对象时总是将它们设置为 NULL。如果不这么做，那么对象将是未定义的，而且对 NULL 的测试甚至不能工作。换句话说，如果不定义对象，那么 if (spriteobj == NULL) 可能会导致程序奔溃，而不是返回 false 结果。

而后可通过调用 D3DXCreateSprite 函数来初始化对象。这个函数实际上是将精灵处理器附着在 Direct3D 设备上，以便它知道如何在后台缓冲区中绘制精灵。

```
HRESULT WINAPI D3DXCreateSprite(
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXSPRITE *ppSprite
);
```

以下是调用这个函数的方法的示例：

```
result = D3DXCreateSprite(d3ddev, &spriteobj);
```

#### 1. 启动精灵渲染器

待会儿就会讲解如何装载精灵图像，但现在先让我们看看如何使用 ID3DXSprite。在主 Direct3D 设备中调用了 BeginScene 之后，就可以开始绘制精灵。首先要做的是锁住表面，以便绘制精灵。这可以通过调用 ID3DXSprite 来实现，其格式如下：

```
HRESULT Begin(
    DWORD Flags
);
```

flags 参数是必需的，其值通常为 D3DXPRITE\_ALPHABLEND，表示绘制精灵时支持透明。以下是一个示例：

```
spriteobj->Begin(D3DXSPRITE_ALPHABLEND);
```

#### 2. 绘制精灵

绘制精灵比仅仅使用源和目标矩形图像做位块传输（如第 5 章讲述的）要复杂一些。不过，D3DXSprite 只使用一个函数 Draw 来处理所有的变换选项，所以一旦理解了这个函数的工作原理，那么只需调整参数就可以执行透明、缩放和旋转。以下是 Draw 函数的声明：

```

HRESULT Draw(
    LPDIRECT3DTEXTURE9 pTexture,
    CONST RECT *pSrcRect,
    CONST D3DXVECTOR3 *pCenter,
    CONST D3DXVECTOR3 *pPosition,
    D3DCOLOR Color
);

```

第一个参数是最重要的一个参数，因为它指定精灵所用的源图像的纹理。第二个参数也很重要，因为可使用它从源图像中抓出“图片单元”，于是就能将精灵的所有动画帧储存在单个位图文件中（本章稍后会有更多介绍）。第三个参数指定旋转发生的中心点。第四个参数指定精灵的位置，通常在这里设置 x 和 y 的值。最后一个参数指定在绘制精灵图像时对其进行的色彩变更（并且不影响透明度）。

**建议** 注意，我们将在第7章学习如何应用矩阵变换来旋转、缩放及平移精灵。我发现先从简单的渲染开始然后再进入像变换这样的复杂主题将更有帮助。因为在进入第7章时你已经对 ID3DXSprite 有了很好的理解。

D3DXVECTOR3 是一个有三个成员变量的 DirectX 数据类型：x、y 和 z。

```

typedef struct D3DXVECTOR3 {
    FLOAT x;
    FLOAT y;
    FLOAT z;
} D3DXVECTOR3;

```

在 2D 屏幕表面移动精灵只需要前两个成员：x 和 y。我们很快就会在一个示例程序中见识如何使用 Draw。

### 3. 停止精灵渲染器

在完成了精灵的绘制之后并且在调用 EndScene 之前，必须调用 ID3DXSprite.End 来对表面解锁，以便其他进程使用。语法如下：

```
HRESULT End(VOID);
```

其用法很直观，因为函数很短：

```
spriteobj->End();
```

### 6.3.2 绘制透明的精灵

ID3DXSprite 并不关心精灵的源图像使用的是颜色键还是 alpha 通道来实现透明，它只是按需渲染图像。如果图像有 alpha 通道，例如一个 32 位的 targa，那么它将以 alpha 进行渲染，包括在图像定义了部分 alpha 范围时和背景进行的半透明混合。但如果图像没有 alpha 通道而使用背景颜色键来制作透明效果，例如 24 位位图，那么颜色键值像素就不会被绘制。甚至可以同时使用 alpha 通道和颜色键来绘制有透明效果的图像。



可以对整个游戏只使用颜色键实现透明，但使用这种技术在质量上会有限制，因为除非执行某种形式的渲染时混合（render-time blending），否则在图像中就必须有离散像素。虽然在运行时进行 alpha 混合是可能的，但这不是开发游戏的好方式，最好应该提前将美工工作准备好。

使用 alpha 通道是渲染透明图像的更好方法（尤其对艺术家来说）。alpha 混合的优势之一是它能够支持部分透明，也就是说可以进行半透明混合。使用 alpha 级实现半透明效果，艺术家可以在精灵的边缘做出混合边界来（对比看来极为出色），而不仅仅是颜色键的精灵所能实现的黑色边界（老学究的突出显示精灵的方法）。要实现这种效果，必须使用支持 32 位 RGBA 图像的文件格式。Targa 是个好选择，PNG 文件也工作良好。我们再来看一下宇宙飞船精灵，这一次使用 alpha 通道而不使用颜色键背景。注意背景中的棋盘格样式，这是在图形编辑器中显示 alpha 通道的常用方式。

现在了解了 D3DXSprite 如何使用 Direct3D 纹理绘制透明精灵（至少理论上理解了！），那么就让我们编写一个简短的程序来实现这一切吧。可以从 CD-ROM 中装载这个项目，也可以修改第 5 章中的 Anim\_Sprite 项目。这是学习的最好方式。

### 1. 创建透明的精灵程序

首先，启动 Visual C++ 并且创建一个新的 Win32 项目，将其命名为 Trans\_Sprite。需要从上一个项目（第 5 章）中复制源代码文件到新的项目中。这些源代码文件是：

- MyGame.cpp
- MyDirectX.h
- MyDirectX.cpp
- MyWindows.cpp

然后可以先编辑 MyGame.cpp 文件并且从这里创建新项目。还有另外一种选择，这也是我推荐的。在 CD-ROM 中，在本章文件夹下，有一个名为 DirectX\_Project 的项目。这是本章的模板项目，在 CD-ROM 的每个章节的文件夹下都有一个同名的项目，其中含有该章所需的所有最新函数和变量。

在阅读本书时可以使用这一项目作为自己的工作模板，而无需打开已完成的项目（在本例中是 Trans\_Sprite，也在 CD-ROM 中）。打开已有的、已经可以运行的项目不会有良好的学习体验！每次都从 DirectX\_Project 作为开始，并按照列出的代码完成剩余的工作，可以学习更多东西。无论使用哪种方式，我都将假设你有一个可工作的、可以运行的 Trans\_Sprite 项目，但这个项目不会带来任何用处。

### 2. 修改 MyDirectX.h

现在需要在名为 MyDirectX.h 的框架文件中加入对纹理装载的支持。这个文件已经在项目中了，只需打开它然后添加新的代码行，让 LoadTexture 函数在整个项目中可见即可。另外，在这里我们也要添加 GetBitmapSize 函数。

在 MyDirectX.h 的 Direct3D 函数原型中加入如下突出显示的代码：

```
LPDIRECT3DTEXTURE9 LoadTexture(string filename,
    D3DCOLOR transcolor = D3DCOLOR_XRGB(0,0,0));
```

在添加了这一行之后，Direct3D 函数的清单如下：

```
//Direct3D functions
bool Direct3D_Init(HWND hwnd, int width, int height, bool fullscreen);
void Direct3D_Shutdown();
LPDIRECT3DSURFACE9 LoadSurface(string filename);
void DrawSurface(LPDIRECT3DSURFACE9 dest, float x, float y,
    LPDIRECT3DSURFACE9 source);

D3DXVECTOR2 GetBitmapSize(string filename);

LPDIRECT3DTEXTURE9 LoadTexture(string filename,
    D3DCOLOR transcolor = D3DCOLOR_XRGB(0,0,0));
```

我们也需要在 MyDirectX.h 文件中加入精灵渲染对象。这个定义可以添加在文件中的任何位置上：

```
extern LPD3DXSPRITE spriteobj;
```

这个新对象被定义为 extern 是因为真正的定义在 MyDirectX.cpp 中。这里的定义仅告诉编译器在代码中碰到 spriteobj 时别紧张，这个对象的确存在。之后，链接器将会找到 MyDirectX.cpp 中的函数。

### 3. 修改 MyDirectX.cpp

既然已经定义了新的 LoadTexture 函数，那么程序的其他部分就都可以使用它了。现在应该打开 MyDirectX.cpp 文件并且在这个文件中添加实际的函数实现。也将在这里添加 GetBitmapSize 函数。

```
LPDIRECT3DTEXTURE9 LoadTexture(std::string filename, D3DCOLOR transcolor)
{
    LPDIRECT3DTEXTURE9 texture = NULL;

    //get width and height from bitmap file
    D3DXIMAGE_INFO info;
    HRESULT result = D3DXGetImageInfoFromFile(filename.c_str(), &info);
    if (result != D3D_OK) return NULL;

    //create the new texture by loading a bitmap image file
    D3DXCreateTextureFromFileEx(
        d3ddev,           //Direct3D device object
        filename.c_str(), //bitmap filename
        info.Width,       //bitmap image width
        info.Height,      //bitmap image height
        1,                //mip-map levels (1 for no chain)
        D3DPOOL_DEFAULT, //the type of surface (standard)
        D3DFMT_UNKNOWN,  //surface format (default)
        D3DPOOL_DEFAULT, //memory class for the texture
        D3DX_DEFAULT,    //image filter
        D3DX_DEFAULT,    //mip filter
        transcolor,      //color key for transparency
        &info,            //bitmap file info (from loaded file)
        NULL,            //color palette
        &texture );       //destination texture

    //make sure the bitmap texture was loaded correctly
```



```

    if (result != D3D_OK) return NULL;

    return texture;
}

D3DXVECTOR2 GetBitmapSize(string filename)
{
    D3DXIMAGE_INFO info;
    D3DXVECTOR2 size = D3DXVECTOR2(0.0f, 0.0f);
    HRESULT result = D3DXGetImageInfoFromFile(filename.c_str(), &info);
    if (result == D3D_OK)
        size = D3DXVECTOR2((float)info.Width, (float)info.Height);
    else
        size = D3DXVECTOR2((float)info.Width, (float)info.Height);

    return size;
}

```

现在我们需要对 Direct3D\_Init 和 Direct3D\_Shutdown 进行修改, 以便在这些函数被调用时自动创建并销毁精灵对象。修改的部分以粗体文本突出显示。

```

bool Direct3D_Init(HWND window, int width, int height, bool fullscreen)
{
    //initialize Direct3D
    d3d = Direct3DCreate9(D3D_SDK_VERSION);
    if (!d3d) return false;

    //set Direct3D presentation parameters
    D3DPRESENT_PARAMETERS d3dpp;
    ZeroMemory(&d3dpp, sizeof(d3dpp));
    d3dpp.hDeviceWindow = window;
    d3dpp.Windowed = (!fullscreen);
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
    d3dpp.EnableAutoDepthStencil = 1;
    d3dpp.AutoDepthStencilFormat = D3DFMT_D24S8;
    d3dpp.Flags = D3DPRESENTFLAG_DISCARD_DEPTHSTENCIL;
    d3dpp.PresentationInterval = D3DPRESENT_INTERVAL_IMMEDIATE;
    d3dpp.BackBufferFormat = D3DFMT_X8R8G8B8;
    d3dpp.BackBufferCount = 1;
    d3dpp.BackBufferWidth = width;
    d3dpp.BackBufferHeight = height;

    //create Direct3D device
    d3d->CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, window,
        D3DCREATE_SOFTWARE_VERTEXPROCESSING, &d3dpp, &d3ddev);
    if (!d3ddev) return false;

    //get a pointer to the back buffer surface
    d3ddev->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, &backbuffer);
    //create sprite object
    D3DXCreateSprite(d3ddev, &spriteobj);

    return 1;
}

```

```

}

void Direct3D_Shutdown()
{
    if (spriteobj) spriteobj->Release();
    if (d3ddev) d3ddev->Release();
    if (d3d) d3d->Release();
}

```

#### 4. 修改 MyGame.cpp

现在新的纹理和精灵功能都已经添加到了支持文件中，下面就可以演示如何绘制有透明效果的精灵了。将以粗体文本突出显示纹理和精灵特定的代码行，以便大家比较这个程序与之前第5章的项目之间的区别。在把支持的框架代码（WinMain、Direct3D\_Init等）从主源代码文件中移出之后，现在的代码是不是简单多了？这对游戏或者演示而言难道不是很棒吗？

```

/*
    Beginning Game Programming, Third Edition
    MyGame.cpp
*/

#include "MyDirectX.h"
using namespace std;

const string APPTITLE = "Transparent Sprite Demo";
const int SCREENW = 1024;
const int SCREENH = 768;

LPDIRECT3DTEXTURE9 image_colorkey = NULL;
LPDIRECT3DTEXTURE9 image_alpha = NULL;
LPDIRECT3DTEXTURE9 image_notrans = NULL;

/**
    ** Game initialization
    **/
bool Game_Init(HWND window)
{
    //initialize Direct3D
    if (!Direct3D_Init(window, SCREENW, SCREENH, false))
    {
        MessageBox(0, "Error initializing Direct3D", "ERROR", 0);
        return false;
    }

    //initialize DirectInput
    if (!DirectInput_Init(window))
    {
        MessageBox(0, "Error initializing DirectInput", "ERROR", 0);
        return false;
    }

    //load non-transparent image

```

```

    image_notrans = LoadTexture("shuttle_notrans.bmp");
    if (!image_notrans) return false;

    //load color-keyed transparent image
    image_colorkey = LoadTexture("shuttle_colorkey.bmp", D3DCOLOR_XRGB(255,0,255));
    if (!image_colorkey) return false;

    //load alpha transparent image
    image_alpha = LoadTexture("shuttle_alpha.tga");
    if (!image_alpha) return false;

    return true;
}

/**
** Game update
**/
void Game_Run(HWND window)
{
    //make sure the Direct3D device is valid
    if (!d3ddev) return;

    //update input devices
    DirectInput_Update();

    //clear the scene
    d3ddev->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
        D3DCOLOR_XRGB(0,0,100), 1.0f, 0);
    //start rendering
    if (d3ddev->BeginScene())
    {
        //start drawing
        spriteobj->Begin(D3DXSPRITE_ALPHABLEND);

        //draw the sprite
        D3DXVECTOR3 pos1( 10, 10, 0);
        spriteobj->Draw( image_notrans, NULL, NULL, &pos1,
            D3DCOLOR_XRGB(255,255,255));

        D3DXVECTOR3 pos2( 350, 10, 0);
        spriteobj->Draw( image_colorkey, NULL, NULL, &pos2,
            D3DCOLOR_XRGB(255,255,255));

        D3DXVECTOR3 pos3( 700, 10, 0);
        spriteobj->Draw( image_alpha, NULL, NULL, &pos3,
            D3DCOLOR_XRGB(255,255,255));

        //stop drawing
        spriteobj->End();

        //stop rendering
        d3ddev->EndScene();
        d3ddev->Present(NULL, NULL, NULL, NULL);
    }
}

```



```

//Escape key ends program
if (KEY_DOWN(VK_ESCAPE)) gameover = true;

//controller Back button also ends
if (controllers[0].wButtons & XINPUT_GAMEPAD_BACK)
    gameover = true;
}

/**
** Game shutdown
**/
void Game_End()
{
    //free memory and shut down
    image_notrans->Release();
    image_colorkey->Release();
    image_alpha->Release();

    DirectInput_Shutdown();
    Direct3D_Shutdown();
}

```

### 5. 运行 Trans\_Sprite 程序

图 6-1 显示了 Trans\_Sprite 程序的运行情况。在程序中绘制了三张相同的图像，因为每一张演示不同的透明类型。

左边第一张本质上没有透明度，因为源位图不包含 alpha 通道也没有合适的透明颜色键。ID3DXSprite::Draw 代码默认使用黑色作为颜色键。在 DirectX 中使用下面这个宏来创建黑色：D3DCOLOR\_XRGB(0,0,0)。

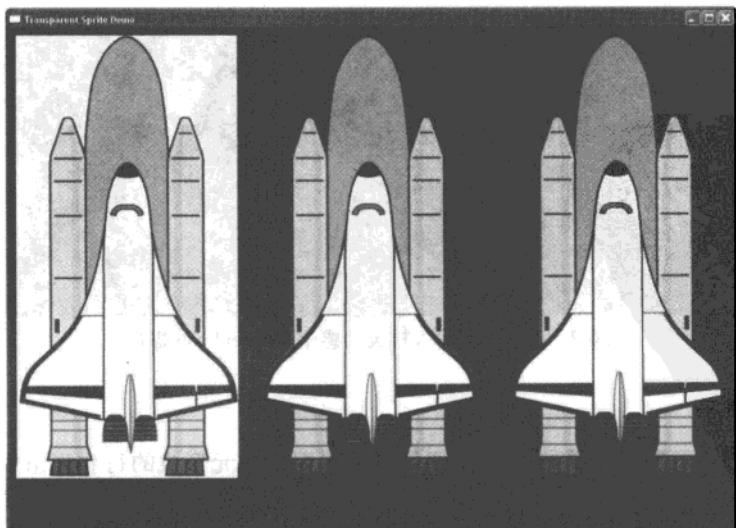


图 6-1 Trans\_Sprite 程序演示如何使用 Direct3D 绘制透明精灵

不过，这里有个问题。如果认真查看第一张航天飞机的图像，可以分辨出图像的黑色部分（在鼻锥和机翼前沿）是用背景颜色绘制的。要亲自运行 `Trans_Sprite` 程序，因为在印刷的书页中蓝色和黑色看起来一样，无法分辨其差别。如果自己运行程序，则肯定会看穿航天飞机，看到其后的蓝色背景。

这是为什么呢？难道我们绘制的这个图像不带透明效果？当然不是！不难看出，在源图像中大多数背景区域是白色的。不过，如果我们使用白色作为透明颜色键，那么航天飞机（它是白色的）的大部分也将变成透明的！

这里的关键是在做美工时对颜色的使用策略。当图像自身有黑色像素时不能使用黑色作为颜色键。在装载纹理时必须使用不同的颜色或者更改颜色键的定义。

接下来，看屏幕截图中的第二张图像。哎呀，这一张才是正确的！在精灵的边缘有透明度，而且内部的黑色区域得到了渲染，它们不是透明的！这正是我们希望的样子。

但是，屏幕截图中的第三张图像有与第一个精灵一样的透明像素问题，值得我们关注，虽然其背景的外沿部分是透明的。这透露了 `ID3DXSprite::Draw` 处理纹理的方式。如果在装载纹理时指定了透明颜色键，则 `ID3DXSprite::Draw` 将使用这一颜色键，即使在图像中还有 `alpha` 通道（而这其实是处理透明像素更好的方法）。

## 6.4 绘制动画的精灵

到目前为止，我们一直是使用单一的位图图像来创建、操纵及绘制精灵。这对于学习精灵编程是良好的起点，但其还不够有效率。例如，如果要使用这种技术来做动画，那么游戏就必须保存许多纹理（一帧一个），这样既慢又乏味。在单一的平铺了图片单元的位图图像中编排动画的各个帧，这种处理动画的方式要好得多。我在第 5 章中对此有过暗示，那时我将一些平铺了图片单元的图像（坦克精灵）展示给了读者。图 6-2 显示了一个奔跑的穴居人角色。

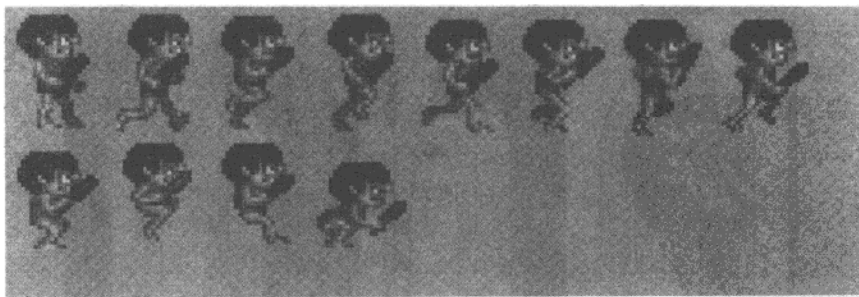


图 6-2 动画的穴居人精灵（感谢 Ari Feldman 提供）

### 6.4.1 使用精灵表

高效地绘制动画的技巧在于理解源图像可以由图片单元形成的行和列组成，在精灵的上下文中，我们称这种平铺的图像为精灵表（`sprite sheet`）。需要做的是，算出图片单元的左上角在位图图像中的位置，然后从图像中按照精灵的宽度和高度复制出一个源矩形来。幸运的是，

ID3DXSprite::Draw 函数可以用来定义图像的源矩形。这使得我们在需要时可以在大图像中指定出一个小的部分，或者说图片单元，或者说帧。想想这是怎么做成的吧，我们有了一种非常快速高效的绘制动画的方法！

图 6-3 显示了一个定义有行和列的爆炸精灵。使用这个图片对照下面的动画算法可帮助我们理解在对动画进行渲染时（一次一帧），从源图像中复制出动画帧的方法。

首先，需要算出图片单元的左位置，或者  $x$ 。这通过使用模运算符  $\%$  来实现。模运算返回除法的余数。例如，如果当前帧是 20，而且在位图中只有 5 个列，那么模运算会给出图片单元的水平起始位置（将它和精灵宽度做乘法）。计算图片单元的顶部边缘只需将当前帧除以列的数量，然后将结果乘以精灵的高度即可。如果有 5 个列，那么图片单元 20 将会在第 4 行、第 5 列。以下是伪码：

```
left = (current frame % number of columns) * sprite width
top = (current frame / number of columns) * sprite height
```

以下是一个示例。注意，在计算中使用了精灵宽度和高度求出源矩形的左边和顶部及右边和底部边缘。

```
left = (curframe % columns) * width;
top = (curframe / columns) * height;
right = left + width;
bottom = top + height;
```

假设在这个示例中，爆炸动画的帧图像精灵表有 6 个列（阔）和 5 个行（深），每一帧尺寸是  $128 \times 128$  像素，我们想绘制第 10 帧。记得，行和列是从 0 开始计数的，C++ 中的数组和序列几乎都是以 0 为基数。所以，如果想绘制第 10 帧，那么在精灵表中要找第 9 个位置的帧（因为从 0 开始数）。

```
left = (9 % 6) * 128; // = 3 * 128 (use remainder)
top = (9 / 6) * 128; // = 1 * 128 (use quotient)
right = left + 128;
bottom = top + 128;
```

答案见图 6-4（不过要先自己算一下）。要检查你是否算得对，从左上角开始数，以从左到右、从上到下顺序，直到到达“第 10”。还要记得的是，它在逻辑上被当成“第 9”。

使用这个技术绘制单个帧，我们可以编写一个能绘制精灵表中任何一帧的函数！为了让这个函数尽可能可重用，必须将能描述精灵的所有属性传递进来，包括源纹理、 $x$  和  $y$  位置、帧号、帧尺寸及列数量等。我们来看一下这个函数：

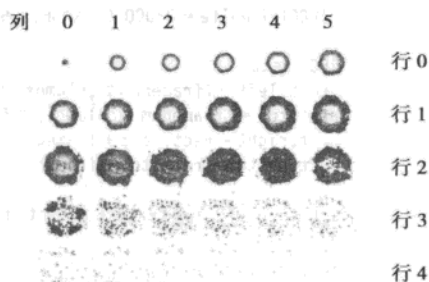


图 6-3 动画的帧以列和行排列



```

void Sprite_Draw_Frame(LPDIRECT3DTEXTURE9 texture, int destx, int desty,
    int framenum, int framew, int frameh, int columns)
{
    D3DXVECTOR3 position( (float)destx, (float)desty, 0 );
    D3DCOLOR white = D3DCOLOR_XRGB(255,255,255);

    RECT rect;
    rect.left = (framenum % columns) * framew;
    rect.top = (framenum / columns) * frameh;
    rect.right = rect.left + framew;
    rect.bottom = rect.top + frameh;

    spriteobj->Draw( texture, &rect, NULL, &position, white);
}

```

作为最后一个参数传递给 ID3DXSprite::Draw 的颜色可以是任何颜色，但白色是最常用的。如果使用不同的颜色，将会导致精灵以这种颜色作为阴影来绘制。甚至可以传递一个带 alpha 成分的颜色让精灵以 alpha 混合的形式淡入淡出！要想实现这一效果，需要将 D3DCOLOR\_XRGB 宏替换为一个不同的宏：D3DCOLOR\_RGBA，它接受 4 个参数（红、绿、蓝和 alpha）。我们后面将使用这个小技巧来做一些特殊效果。

此外，只要有创意地使用定时器，就可以设计出一个以任何其他的帧速率来显示动画的函数。GetTickCount() 函数（由 Windows API 提供）返回的毫秒值可用于计时。只要以引用方式传递一个帧变量和一个起始时间变量，这个函数就可以修改这些引用变量值。于是，当每次调用这个函数时，帧号和定时器都会得到更新。将 Sprite\_Animate 和 Sprite\_Draw\_Frame 一起使用，可以提供高质量精灵动画和渲染所需的一切。

```

void Sprite_Animate(int &frame, int startframe, int endframe,
    int direction, int &starttime, int delay)
{
    if ((int)GetTickCount() > starttime + delay)
    {
        starttime = GetTickCount();

        frame += direction;
        if (frame > endframe) frame = startframe;
        if (frame < startframe) frame = endframe;
    }
}

```

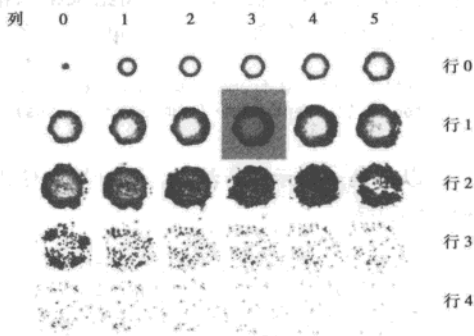


图 6-4 在精灵表中，第 10 动画帧是突出显示的



## 6.4.2 精灵动画演示

我们肯定需要一个示例来正确地演示精灵动画。学了这么多理论，我不知道你会有什么想法，但我是需要写一些代码的。我们就从在框架中添加新的助手函数来开始吧。这里假定你可以自己创建新项目或者在创建新程序时使用 DirectX\_Project 模板项目作为起点了。如果尚不清楚创建及配置项目的方法，可参阅附录 A，其中有对整个过程的彻底的讲解。

### 1. 更新 MyDirectX.h

这个在将新的功能添加到我们的游戏编程工具箱中时，更新框架文件的过程，将会很快成为我们的习惯动作，因为我们太经常做这件事情了。下面我只把新函数列出来，你可自行将它添加到框架中。以后添加自己的可重用函数就会成为我们的习惯之举，而很快你也会有一个全功能的游戏库了。

我们以之前创建的两个新精灵动画函数作为开始。这两个函数所需的原型定义需要添加到 MyDirectX.h 头文件中。它们是：

```
//new prototype functions added to MyDirectX.h file
void Sprite_Draw_Frame(LPDIRECT3DTEXTURE9 texture, int destx, int desty,
    int framenum, int framew, int frameh, int columns);
void Sprite_Animate(int &frame, int startframe, int endframe,
    int direction, int &starttime, int delay);
```

**建议** 注意，本章出现的新更改没有包括在 DirectX\_Project 模板项目中。它将保留第 5 章介绍的代码，却不带新代码。这么做是为了让它成为一个工作区，让读者自己可以实现每个新章节中的新代码。如果想使用本项目完全完成的版本，可往前跳到第 7 章，它里面也含有所有本章的新代码。最终版将包括在 CD-ROM 的 \sources\DirectX\_Project 文件夹中。

### 2. 更新 MyDirectX.cpp

这两个函数的完整实现必须添加到 MyDirectX.cpp 文件中。这两个函数位于文件中的什么位置实在是不重要，但我建议将它们放在 LoadSurface 函数之下。

```
void Sprite_Draw_Frame(LPDIRECT3DTEXTURE9 texture, int destx, int desty,
    int framenum, int framew, int frameh, int columns)
{
    D3DXVECTOR3 position( (float)destx, (float)desty, 0 );
    D3DCOLOR white = D3DCOLOR_XRGB(255,255,255);

    RECT rect;
    rect.left = (framenum % columns) * framew;
    rect.top = (framenum / columns) * frameh;
    rect.right = rect.left + framew;
    rect.bottom = rect.top + frameh;

    spriteobj->Draw( texture, &rect, NULL, &position, white);
}
```

```
void Sprite_Animate(int &frame, int startframe, int endframe,
```

```

    int direction, int &starttime, int delay)
{
    if ((int)GetTickCount() > starttime + delay)
    {
        starttime = GetTickCount();

        frame += direction;
        if (frame > endframe) frame = startframe;
        if (frame < startframe) frame = endframe;
    }
}

```

### 3. 动画精灵源代码

名称为 `Animate_Sprite` 的实际的演示源代码，如同往常一样，位于 `MyGame.cpp` 中。由于我们已经多次见到类似的代码了，所以后面将开始减少代码清单中的注释，而你现在应该对这些代码很熟悉了。该程序的运行如图 6-5 所示。

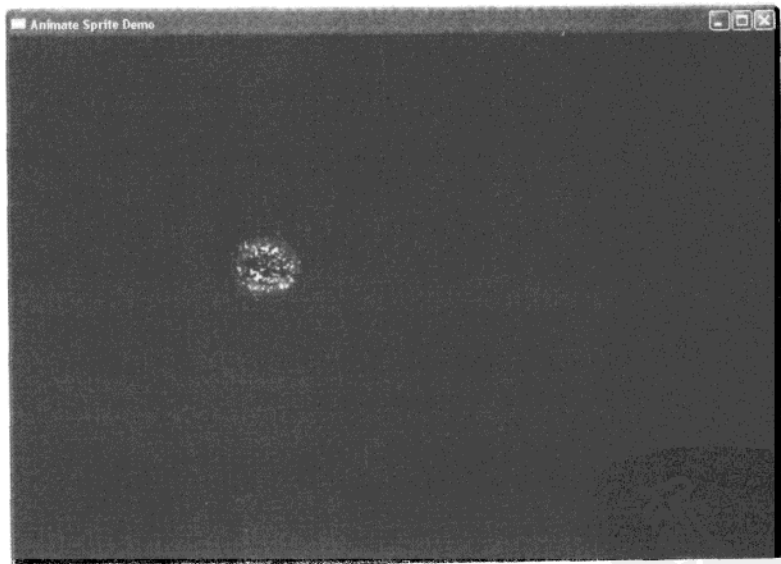


图 6-5 `Animate_Sprite` 程序演示动画

```

#include "MyDirectX.h"
using namespace std;

const string APPTITLE = "Animate Sprite Demo";
const int SCREENW = 800;
const int SCREENH = 600;

LPDIRECT3DTEXTURE9 explosion = NULL;
int frame = 0;

```

```
int starttime = 0;

bool Game_Init(HWND window)
{
    if (!Direct3D_Init(window, SCREENW, SCREENH, false))
    {
        MessageBox(0, "Error initializing Direct3D", "ERROR", 0);
        return false;
    }
    if (!DirectInput_Init(window))
    {
        MessageBox(0, "Error initializing DirectInput", "ERROR", 0);
        return false;
    }

    //load explosion sprite
    explosion = LoadTexture("explosion_30_128.tga");
    if (!explosion) return false;
    return true;
}

void Game_Run(HWND window)
{
    if (!d3ddev) return;

    DirectInput_Update();

    d3ddev->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
        D3DCOLOR_XRGB(0,0,100), 1.0f, 0);

    if (d3ddev->BeginScene())
    {
        //start drawing
        spriteobj->Begin(D3DXSPRITE_ALPHABLEND);
        //animate and draw the sprite
        Sprite_Animate(frame, 0, 29, 1, starttime, 30);
        Sprite_Draw_Frame(explosion, 200, 200, frame, 128, 128, 6);

        //stop drawing
        spriteobj->End();

        d3ddev->EndScene();
        d3ddev->Present(NULL, NULL, NULL, NULL);
    }

    //exit with Escape key or controller Back button
    if (KEY_DOWN(VK_ESCAPE)) gameover = true;
    if (controllers[0].wButtons & XINPUT_GAMEPAD_BACK)
        gameover = true;
}

void Game_End()
{
    explosion->Release();
}
```

```
DirectInput_Shutdown();  
Direct3D_Shutdown();  
}
```

## 6.5 你所学到的

本章我们学习了使用 D3DXSprite 在 Direct3D 中绘制透明精灵的方法。有以下几个要点：

- 如何创建 D3DXSprite 对象。
- 如何从位图文件装载纹理。
- 如何绘制透明精灵。
- 如何从单个位图中抓取精灵动画帧。

## 6.6 复习测验

以下这些复习题可考察读者从本章学到了多少东西。

- 1) 用于处理精灵的 DirectX 对象的名称是什么？
- 2) 将位图图像装载到纹理对象中的函数是什么？
- 3) 用于创建精灵对象的函数是什么？
- 4) 绘制精灵的 D3DX 函数的名称是什么？
- 5) D3DX 纹理对象的名称是什么？
- 6) 哪个函数返回位图文件中图像的尺寸？
- 7) 哪个 Windows API 函数提供用于计时的时钟滴答值？
- 8) 在绘制任何精灵之前必须要调用的函数的名称是什么？
- 9) 在精灵绘制完成后要调用的函数的名称是什么？
- 10) 在精灵绘制函数中用于指定源矩形的数据类型是什么？

## 6.7 自己动手

以下练习可考查你对本章中的内容的掌握程度。

习题 1 本章的 AnimateSprite 项目的确展示了动画的工作原理，但它却是个有点乏味的演示。修改程序，让它在同一个时间内生成不止一个爆炸精灵。

习题 2 以习题 1 为基础，修改此项目，让每个爆炸动画以随机的动画速率显示。

## 第7章 精灵变换

本章继续我们从第6章开始的对精灵编程的学习。这一章我们将给我们的游戏编程工具箱添加变换精灵的能力，而不仅仅是绘制带与不带动画的精灵，从而可以使用 Direct3D 矩阵及一个极为方便的 D3DX 助手函数来旋转、缩放及平移精灵。在编写这些精灵函数时，你可能会想到将所有功能合并到一个大的精灵绘制函数中，或者合并到一个精灵类中。虽然我可以鼓励创建一个 C++ 类来处理精灵，但却不推荐将许多函数合并成一个大函数。你可记得，我们可以重载同名函数，并通过不同的参数集来区分；而且我们可以设置参数的默认值。让我们好好利用 C++ 语言的这些伟大的特性，并且有创造性地使用自定义数据类型，从而有效地处理精灵吧。

本章将学到：

- 如何旋转精灵。
- 如何缩放精灵。
- 如何平移精灵。
- 如何对 2D 图形做矩阵变换。
- 如何以动画方式绘制变换的精灵。

### 7.1 精灵旋转和缩放

由于有 D3DX 库，精灵的旋转和缩放实现起来相对简单。不管是要绘制来自单个精灵表中的单帧的精灵，还是进行复杂的动画，我们都可使用同样的多功能的 ID3DXSprite::Draw() 函数。不过，为了增加这些特殊功能，需要多加一个步骤。还记得上一章中的 Sprite\_Draw\_Frame() 函数吧，它是这样的：

```
void Sprite_Draw_Frame(LPDIRECT3DTEXTURE9 texture, int destx, int desty,
    int framenum, int framew, int frameh, int columns)
{
    D3DXVECTOR3 position( (float)destx, (float)desty, 0 );
    D3DCOLOR white = D3DCOLOR_XRGB(255,255,255);
    RECT rect;
    rect.left = (framenum % columns) * framew;
    rect.top = (framenum / columns) * frameh;
    rect.right = rect.left + framew;
    rect.bottom = rect.top + frameh;
    spriteobj->Draw( texture, &rect, NULL, &position, white);
}
```

这个函数可很好地处理动画，要是不考虑任何特殊效果（例如旋转和缩放）那么它也是个很好的通用精灵动画函数。不过我们的确关心特殊效果！当使用默认参数值定义函数原型时，函数可很容易地绘制简单的非动画精灵。注意以下的函数定义：

```
void Sprite_Draw_Frame(
    LPDIRECT3DTEXTURE9 texture,
    int destx,
    int desty,
    int framenum,
    int framew,
    int frameh,
    int columns
);
```

如果可以重新对函数参数做一点编排，那么，在需要绘制单帧精灵的情况中（例如没有动画帧），我们可对 framenum 和 columns 参数使用默认值：

```
void Sprite_Draw_Frame(
    LPDIRECT3DTEXTURE9 texture = NULL,
    int destx = 0,
    int desty = 0,
    int framew = 64,
    int frameh = 64,
    int framenum = 0,
    int columns = 1
);
```

对于这个函数定义，需要更改该函数的实现以便与之匹配。注意，函数实现中的参数无需包含默认值，它们只需在函数原型中定义即可（会在 MyDirectX.h 头文件中定义）。在调用函数时，任何带有默认值的参数都可以跳过；不过，如果忽略一个参数，那么就必須忽略该参数后面的所有参数。由于这个原因，我们必须把最为不定的参数放在参数列表的最后，而最为重要或不可选的参数则放在参数列表的前面。例如在下面的函数中，纹理参数绝对是重要的，因为没有它什么都画不出来。destx 和 desty 参数也是重要的，但如果让图像绘制在默认位置 (0,0) 也是可行的。帧的 width 和 height 参数也是重要的，但给出 (64,64) 作为它们的默认值，因为这是游戏精灵中所用的非常常见的尺寸。最后两个参数是最常用做可选的参数，尤其对于非动画精灵（例如位图文件中的单个图像），它们的默认值分别为 0 和 1，表示对单一帧图像进行渲染。

还有一个细节，由于纹理参数如果不提供（虽然这种情况似乎不太可能），其默认值将是 NULL，因此必须在尝试绘制图像之前执行检查，看是否为 NULL。这是个好做法，建议在处理诸如 LPDIRECT3DTEXTURE9 或者任何为空时会导致崩溃的指针时，都采取这样的做法。

```
void Sprite_Draw_Frame(
    LPDIRECT3DTEXTURE9 texture,
    int destx,
    int desty,
    int framew,
    int frameh,
    int framenum,
    int columns)
{
    //perform check for NULL
    if (!texture) return;
```

```

D3DXVECTOR3 position( (float)destx, (float)desty, 0 );
D3DCOLOR white = D3DCOLOR_XRGB(255,255,255);
RECT rect;
rect.left = (framenum % columns) * framew;
rect.top = (framenum / columns) * frameh;
rect.right = rect.left + framew;
rect.bottom = rect.top + frameh;
spriteobj->Draw( texture, &rect, NULL, &position, white);
}

```

保持 `Sprite_Draw_Frame` 原来在 `MyDirectX.h` 和 `MyDirectX.cpp` 中的样子，不过，如果觉得有用的话，鼓励读者进行这些更改。

### 7.1.1 2D 变换

ID3DXSprite 可以处理变换矩阵是它的迷人之处，这让它就如 Direct3D 设备一样！虽然我们尚未深入研究 3D，但我们很快就会进入这一主题，将学习创建视图矩阵、投影矩阵和变换矩阵（通常称为世界矩阵——world matrix）的方法，从而在 3D 场景中渲染对象。ID3DXSprite 的伟大之处在于它可以让我们对精灵做完全相同的事情，只是少了一维（确切地说是 Z 轴）而已。

**建议** 矩阵是个  $4 \times 4$  的阵列，4 行 4 列。它通过非常快速的矩阵数学计算，而不是慢得多的正弦和余弦计算（这是从前在计算机上变换和渲染 3D 图形的方法）来变换要渲染的对象。DirectX 中矩阵的名称是 D3DXMATRIX。

现在，介绍完成这些功能的变换函数。它的名称为 `D3DXMatrixTransformation2D`，其定义如下：

```

D3DXMATRIX * D3DXMatrixTransformation2D(
    D3DXMATRIX * pOut,
    CONST D3DXVECTOR2 * pScalingCenter,
    FLOAT pScalingRotation,
    CONST D3DXVECTOR2 * pScaling,
    CONST D3DXVECTOR2 * pRotationCenter,
    FLOAT Rotation,
    CONST D3DXVECTOR2 * pTranslation
);

```

首先，这个函数生成一个实际上通过引用来传递的矩阵（第一个参数），然后填满矩阵值返回给用户。这个  $4 \times 4$  矩阵包含所有的变换组合：旋转、缩放及平移（例如，“位置”）。我们将在后面学习更多与这些变换有关的知识，目前，我们只需关注变换如何用于操纵精灵即可。

**建议** 在 3D 空间中 3D 向量既可用位置 (x,y,z) 也可以用方向（也是 (x,y,z) 集）来表示。由于只使用 x 和 y 值来进行精灵渲染，因此我们可以使用更简单的 2D 向量进行精灵编程。DirectX 的精灵向量的名称是 D3DXVECTOR2。

#### 1. 变换了的 2D 矩阵

首先，我们来看看传递的、会被填上矩阵数据的参数：



```
D3DMATRIX mat;
```

在这里，D3DMATRIX 是 D3DX 扩展库中一个结构的名称，它由 16 个浮点数组成，排成 4 行 4 列。矩阵不是 Direct3D 独有的，也不是 3D 计算机图形独有的。它是一种数学结构，在大学一年级的代数课（线性代数）上会学习到它。

和所有相对高级的主题一样，不要试着一口气把它吞下，给自己时间，在实践中边用边学习新的概念。我们无需一头钻进 3D 图形的每个概念和函数，以为这样才能有效地使用它们。这种错误是初学者常犯的——试着彻底理解所有的一切，然后因为信息过多而沮丧、不知所措。人类大脑的学习方式不是这样的。我喜欢以这种方式来思考学习——必须要向自己的大脑证实说，需要这些知识是为了生存。似乎有点奇怪，但这是我们运转的方式。如果想身体强健，就必须说服自己的肌肉通过锻炼来增加力量。就如身体的肌肉那样，如果头脑不需要记忆某些东西就不会记忆它。我们的心智，和我们的身体一样，能够也的确会适应我们的环境。不过高级概念的思考，包括计算机编程，是极为抽象的，必须多多练习才行。这就是这个领域既困难又很有回报的原因。

## 2. 精灵缩放

接下来，我们需要为精灵的变换创建一个代表水平和垂直缩放值的向量。使用 D3DXVECTOR2 变量，它可以这样声明：

```
D3DXVECTOR2 scale( 2.0f, 2.0f);
```

在这里，缩放因子会使精灵以其正常尺寸的两倍绘制。下一步，需要定义另外一个向量表示精灵的中心。它会成为旋转的精灵的支点。如果不将支点设置为精灵的中心，那么在试着旋转它的时候，它就会有点像是沿着左上角 (0,0) 旋转而不是沿着中心旋转。

```
D3DXVECTOR2 center( (width*scaling) / 2.0f, (height*scaling) / 2.0f);
```

在这个示例中可以看到，宽度和高度乘上了缩放因子，然后被 2 除（这是为了得到中心点）。如果根本不更改精灵的比例，那么可以不管缩放。但为了完善起见，这里包含了缩放以便在既需要旋转又需要缩放的时候，它能够正确绘制。否则，要是没有缩放因子的话，旋转会有畸变<sup>⊖</sup>。

如果需要，可以在水平轴和垂直轴上使用不同的宽度和高度比例值按不同的比例缩放精灵。由于这种做法不常见，因此在代码中只使用了相同的宽度和高度缩放因子。

## 3. 精灵旋转

旋转是个相当直观的浮点值，不像缩放和平移是个向量。在旋转精灵时有个重要的问题需要记住，旋转角是以角度表示还是弧度表示？

我们马上要用来创建完整变换的精灵矩阵的函数需要的是弧度而不是角度。我们最为熟悉的是范围在 0~359° 之间形成一个圆的角度（例如，指南针的读数就是角度）。但在图形编程中所用

---

⊖ 这里是指不使用缩放因子求出中心坐标就进行旋转的话，由于旋转的中心点不在 sprite 的中心位置上，因此无法得出正确的旋转效果。——译者注

的旋转则需要弧度，其范围是  $0.0 \sim 2.0\pi$ ——因为圆的周长是 2 倍的圆周率乘上半径。如果想要的是只是向量或方向，那么半径可以省掉。

这个公式还有一个更简单的表示：将圆的直径（ $2r$ ）乘以  $\pi$ ， $\pi$  大约等于 3.1415926535。 $\pi$  的两倍大约是 6.28。

我跑题了。我们不需要圆的周长，这只是使用弧度的一个例子。我们所需的是一个在角度和弧度之间转换的函数。在代码中大多数时候我们使用角度，然后在最后一刻需要变换精灵时将角度转换为弧度。从角度转换为弧度的方法是将角度乘以  $\pi$  然后除以 180：

```
radian angle = degree angle * PI / 180
```

我们用一个容易辨识的值来测试这个公式。如果完整的圆有  $2\pi$  弧度，那么  $180^\circ$  就是这个值的一半，也就是  $\pi$ （而不是  $2\pi$ ），也就是  $\pi$  本身。所以，半圆用弧度表示就是 3.14。

```
radian angle = 180 * 3.14 / 180
radian angle = 3.14
```

将这个公式编写到一个可重用的函数中：

```
double toRadians(double degrees)
{
    return degrees * 3.1415926535 / 180.0;
}
```

这段代码只有一个问题。你是否能看出这个函数有低效问题？“ $\pi$ ”不会改变，而“180.0”也不会变。如果在游戏中需要频繁地做这件事情（这是有可能的，因为在典型的游戏中对象的确经常移动和旋转），那么这段代码就会（很愚蠢地）一次又一次地、没有目的地计算这些值。我们需要对其优化。最好的方式就是如下这样预先计算出这些值：

```
const double PI = 3.1415926535;
const double PI_under_180 = 180.0f / PI;
const double PI_over_180 = PI / 180.0f;
```

以下是优化后的函数：

```
double toRadians(double degrees)
{
    return degrees * PI_over_180;
}
```

从弧度转换为角度不常用，但为了更完整，以下也给出这个函数：

```
double toDegrees(double radians)
{
    return radians * PI_under_180;
}
```

#### 4. 精灵平移

需要创建的最后一个参数是平移向量。

```
D3DXVECTOR2 trans( x, y );
```

将平移向量设置为精灵的 (x,y) 位置之后, 在调用 D3DXMatrixTransformation2D 函数以所有这些参数构建矩阵时, 它们就会被编码到矩阵中。接下来就来做这件事:

```
D3DXMatrixTransformation2D(
    &mat,        //the resulting matrix
    NULL,        //scaling center point (not used)
    0,           //scaling rotation value (not used)
    &scale,       //scaling vector
    &center,      //rotation center/pivot vector
    rotation,    //rotation angle
    &trans       //translation vector
);
```

注意, 有些参数是以引用方式传递的, 这也是它们的名称之前有“&”符号的原因。特别地, 函数通过第一个参数返回填满了数据的矩阵的 16 个值。这些值表示了精灵的旋转、缩放和平移变换。有了这个矩阵, 我们就可以告诉精灵对象 (ID3DXSprite) 依据它来作为当前的变换:

```
spriteobj->SetTransform( &mat );
```

## 5. 创建精灵矩阵

D3DX 库在一个函数调用中就为我们做好了一切, D3DXMatrixTransformation2D 构建矩阵; ID3DXSprite::SetTransform 在渲染下一个精灵时使用这个矩阵。下面是函数调用的过程:

```
//create a scale vector
D3DXVECTOR2 scale( scaling, scaling );

//create a translate vector
D3DXVECTOR2 trans( x, y );

//set center by dividing width and height by two
D3DXVECTOR2 center( (float)( width * scaling )/2, (float)( height * scaling )/2);

//create 2D transformation matrix
D3DXMATRIX mat;
D3DXMatrixTransformation2D( &mat, NULL, 0, &scale, &center, rotation, &trans );

//tell sprite object to use the transform
spriteobj->SetTransform( &mat );
```

### 7.1.2 绘制变换了的精灵

剩下的工作就是绘制精灵了! 不过, 与第 6 章不一样, 调用 ID3DXSprite::Draw 时不能使用位置参数, 而是传递 NULL 给它们。动画代码和以前一样:

```
int fx = (frame % columns) * width;
int fy = (frame / columns) * height;
RECT srcRect = {fx, fy, fx + width, fy + height};
```

但是 Draw 函数调用将有一点小变化, 因为要用新的变换矩阵处理精灵的位置:

```
spriteobj->Draw( image, &srcRect, NULL, NULL, color );
```

有了用于变换并绘制高级精灵的代码之后，就可以将这些代码放到 MyDirectX.cpp 文件中作为一个可重用函数了（函数原型名称放在 MyDirectX.h 头文件中）。以下是添加到 MyDirectX.h 中的原型（注意，有些参数有默认值，这就意味着如果不指定这些参数的话函数将使用默认值）：

```
void Sprite_Transform_Draw(
    LPDIRECT3DTEXTURE9 image,
    int x,
    int y,
    int width,
    int height,
    int frame = 0,
    int columns = 1,
    float rotation = 0.0f,
    float scaling = 1.0f,
    D3DCOLOR color = D3DCOLOR_XRGB(255,255,255)
);
```

以下是需要添加到 MyDirectX.cpp 中的新函数（注意，函数的实现不包括默认参数值，默认值只需在原型中出现）：

```
void Sprite_Transform_Draw(LPDIRECT3DTEXTURE9 image, int x, int y,
    int width, int height, int frame, int columns,
    float rotation, float scaling, D3DCOLOR color)
{
    //create a scale vector
    D3DXVECTOR2 scale( scaling, scaling );

    //create a translate vector
    D3DXVECTOR2 trans( x, y );

    //set center by dividing width and height by two
    D3DXVECTOR2 center( (float)( width * scaling )/2, (float)( height * scaling )/2 );

    //create 2D transformation matrix
    D3DXMATRIX mat;
    D3DXMatrixTransformation2D( &mat, NULL, 0, &scale, &center, rotation, &trans );

    //tell sprite object to use the transform
    spriteobj->SetTransform( &mat );

    //calculate frame location in source image
    int fx = (frame % columns) * width;
    int fy = (frame / columns) * height;
    RECT srcRect = {fx, fy, fx + width, fy + height};

    //draw the sprite frame
    spriteobj->Draw( image, &srcRect, NULL, NULL, color );
}
```

### 7.1.3 Rotate\_Scale\_Demo 程序

下面我们看看这个新的可重用函数的作用。以下是一个名为 Rotate\_Scale\_Demo 的示例，其截屏图见图 7-1。

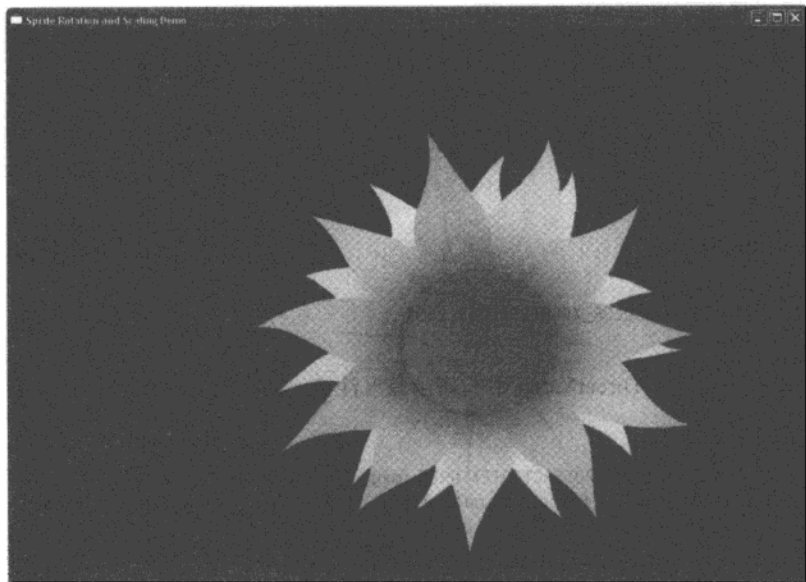


图 7-1 Rotate\_Scale\_Demo 程序绘制带有变换的精灵

```

/*
    Beginning Game Programming, Third Edition
    MyGame.cpp
*/
#include "MyDirectX.h"
using namespace std;

const string APPTITLE = "Sprite Rotation and Scaling Demo";
const int SCREENW = 1024;
const int SCREENH = 768;

LPDIRECT3DTEXTURE9 sunflower;
D3DCOLOR color;
int frame=0, columns, width, height;
int startframe, endframe, starttime=0, delay;

bool Game_Init(HWND window)
{
    //initialize Direct3D
    Direct3D_Init(window, SCREENW, SCREENH, false);

    //initialize DirectInput

```



```

DirectInput_Init(window);

//load the sprite image
sunflower = LoadTexture("sunflower.bmp");

return true;
}

void Game_Run(HWND window)
{
    static float scale = 0.001f;
    static float r = 0;
    static float s = 1.0f;

    //make sure the Direct3D device is valid
    if (!d3ddev) return;

    //update input devices
    DirectInput_Update();

    //clear the scene
    d3ddev->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
        D3DCOLOR_XRGB(0,0,100), 1.0f, 0);

    //start rendering
    if (d3ddev->BeginScene())
    {
        //begin sprite rendering
        spriteobj->Begin(D3DXSPRITE_ALPHABLEND);

        //set rotation and scaling
        r = timeGetTime() / 600.0f;
        s += scale;
        if (s < 0.1 || s > 1.25f) scale *= -1;

        //draw sprite
        width = height = 512;
        frame = 0;
        columns = 1;
        color = D3DCOLOR_XRGB(255,255,255);
        Sprite_Transform_Draw( sunflower, 300, 150, width, height,
            frame, columns, r, s, color );

        //end sprite rendering
        spriteobj->End();

        //stop rendering
        d3ddev->EndScene();
        d3ddev->Present(NULL, NULL, NULL, NULL);
    }

    //exit when escape key is pressed
    if (KEY_DOWN(VK_ESCAPE)) gameover = true;
}

```

```

        //controller Back button also ends
        if (controllers[0].wButtons & XINPUT_GAMEPAD_BACK)
            gameover = true;
    }

    void Game_End()
    {
        //free memory and shut down
        sunflower->Release();

        DirectInput_Shutdown();
        Direct3D_Shutdown();
    }

```

Sprite\_Transform\_Draw() 函数有许多默认参数,在任何无需使用高级功能时,我们就可以利用这一特性来简化代码。比如,如果只需绘制一个简单的(非动画的)精灵,那么在调用函数时就无需提供帧号、列数、旋转、缩放或颜色参数,例如:

```
Sprite_Transform_Draw( spaceship, 100, 100, 64, 64 );
```

函数调用的结果将在(x,y)位置为100 100的地方绘制一个简单精灵图像,它指定精灵的尺寸为64×64像素。我们本来可进一步在Sprite\_Transform\_Draw函数中取出图像的宽度和高度,但这样会增加函数的复杂性,更不用说还降低了速度。

#### 7.1.4 带有变换的动画

旋转等变换功能确实很棒,但如果我们也想制作动画该如何做呢?毕竟,如果精灵不支持储存于精灵表中的动画的话那就没什么用了。幸运的是,我们也可以将新的变换功能应用到动画上。由于ID3DXSprite用于绘制单个或多个帧的精灵,因此我们可以使用相同的变换来旋转或缩放精灵,无论它是否是动画的。图7-2显示了包含动画游戏角色帧的精灵表。

刚刚添加的Sprite\_Transform\_Draw()函数是独立的而且是全功能的,这是这个函数的伟大之处!我们来看这个函数都能做什么:

- 在任意(x,y)位置上绘制简单的(非动画的)精灵
- 旋转一个简单的精灵
- 缩放一个简单的精灵
- 旋转、缩放并平移一个简单的精灵
- 在任意(x,y)位置上绘制动画精灵
- 旋转一个动画精灵
- 缩放一个动画精灵
- 旋转、缩放并平移动画精灵

无论想做什么,这个神奇的函数都能帮你做成。

我们来看看在有动画介入时这个函数如何工作。这里给出一个新的名为Rotate\_Animate\_Demo的程序。这个程序使用在上一个实例中的“向日葵”精灵上使用的同样变换,不同之处在

于要处理动画精灵。有什么不同呢？就 Direct3D 而言，没有不同。Sprite\_Translate\_Draw() 函数在所有的变换都启用的情况下既能处理简单的精灵也能处理动画的精灵。



图 7-2 带有 64 帧动画的角色

我们来试试带有完整变换的动画。图 7-3 显示了 Rotate\_Animate\_Demo 程序的输出，而后列出了其代码。



图 7-3 Rotate\_Animate\_Demo 程序绘制一个变换的动画精灵



```
/*
    Beginning Game Programming, Third Edition
    MyGame.cpp
*/

#include "MyDirectX.h"
using namespace std;

const string APPTITLE = "Sprite Rotation and Animation Demo";
const int SCREENW = 1024;
const int SCREENH = 768;

LPDIRECT3DTEXTURE9 paladin = NULL;
D3DCOLOR color = D3DCOLOR_XRGB(255,255,255);
float scale = 0.004f;
float r = 0;
float s = 1.0f;
int frame=0, columns, width, height;
int startframe, endframe, starttime=0, delay;

bool Game_Init(HWND window)
{
    //initialize Direct3D
    Direct3D_Init(window, SCREENW, SCREENH, false);

    //initialize DirectInput
    DirectInput_Init(window);

    //load the sprite sheet
    paladin = LoadTexture("paladin_walk.png");
    if (!paladin) {
        MessageBox(window, "Error loading sprite", "Error", 0);
        return false;
    }

    return true;
}

void Game_Run(HWND window)
{
    //make sure the Direct3D device is valid
    if (!d3ddev) return;

    //update input devices
    DirectInput_Update();

    //clear the scene
    d3ddev->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
        D3DCOLOR_XRGB(0,0,100), 1.0f, 0);

    //start rendering
    if (d3ddev->BeginScene())
    {
        //begin sprite rendering
```

```

    spriteobj->Begin(D3DXSPRITE_ALPHABLEND);

    //scale the sprite from tiny to huge over time
    s += scale;
    if (s < 0.5f || s > 6.0f) scale *= -1;

    //set animation properties
    columns = 8;
    width = height = 96;
    startframe = 24;
    endframe = 31;
    delay = 90;
    Sprite_Animate(frame, startframe, endframe, 1, starttime, delay);

    //transform and draw sprite
    Sprite_Transform_Draw( paladin, 300, 200, width, height,
        frame, columns, 0, s, color );

    //end sprite rendering
    spriteobj->End();

    //stop rendering
    d3ddev->EndScene();
    d3ddev->Present(NULL, NULL, NULL, NULL);
}

//exit when escape key is pressed
if (KEY_DOWN(VK_ESCAPE)) gameover = true;

//controller Back button also ends
if (controllers[0].wButtons & XINPUT_GAMEPAD_BACK)
    gameover = true;
}

void Game_End()
{
    //free memory and shut down
    paladin->Release();

    DirectInput_Shutdown();
    Direct3D_Shutdown();
}

```

## 7.2 你所学到的

本章我们学习了在单个函数——Sprite\_Transform\_Draw 中，使用支持动画、旋转、缩放、移动和 alpha 混合功能的完整的基于矩阵的变换来渲染 2D 精灵。这个函数将成为我们在将来编写任何 2D 演示和游戏的干将。第 8 章我们将创建新的精灵结构并且学习检测精灵碰撞的两种技术。

概括地说，本章我们学习了如何：

- 执行基于矩阵的精灵缩放。
- 执行基于矩阵的精灵旋转。
- 执行基于矩阵的精灵平移（或移动）。
- 使用“精灵表”图像来实现精灵动画。
- 使用计时技术来设置动画的速度。

### 7.3 复习测验

以下这些复习测验题可用于检验你从本章学到了哪些东西。这些复习测验题的答案在附录 C 中。

- 1) 精灵的源图像使用哪种类型的 Direct3D 对象来处理？
- 2) 使用传递给函数的旋转、缩放和平移向量来创建变换 2D 精灵的矩阵的函数是哪个？
- 3) 在旋转精灵时，角是如何编码的，是角度还是弧度？
- 4) 保存用于精灵缩放向量的数据类型是什么？
- 5) 保存用于精灵移动向量的数据类型是什么？
- 6) 保存用于精灵旋转向量的数据类型是什么？
- 7) 将矩阵应用于精灵变换的 ID3DXSprite 函数是哪个？
- 8) 哪个参数总是需要传递给 ID3DXSprite::Begin 函数？
- 9) 除了宽度、高度和帧号以外，动画还需要哪些值？
- 10) 用于将 alpha 颜色成分编码到 D3DCOLOR 中的是哪个宏？

### 7.4 自己动手

以下练习可考验你对本章中的内容的掌握程度。

- 习题 1 修改旋转动画演示，使用你自己获得的动画精灵（可使用 SpriteLib）替换移动的角色。
- 习题 2 修改旋转缩放演示，当缩放比例改变时，让花朵精灵保持在屏幕中央。



## 第 8 章 检测精灵碰撞

到目前为止我们学习了如何在屏幕上绘制精灵，不过要制作游戏，仅仅具备绘图的能力是远远不够的。这还只是开始而已，就如启动汽车的发动机一样，一旦启动了，就可以开走了。真正的游戏有许多精灵（或者 3D 网格，很快会介绍）相互交互，例如子弹和火箭击中敌人的飞船造成爆炸，必须在迷宫中穿行但不能穿越墙壁的精灵，以及越过板条箱并且落在敌人角色顶上的精灵（就如在《Super Mario World》中 Mario 跳到乌龟上面把它打掉）。所有这些都要求我们能够探测两个精灵的碰撞。碰撞检测是可集成到游戏中的最简单（也是最重要）的物理（physics）类型。精灵碰撞打开了游戏编程的世界并使得构建一个真正的游戏成为可能！

本章将学到：

- 边界框（bounding box）碰撞检测。
- 基于距离的碰撞检测。

### 8.1 边界框碰撞检测

检测精灵碰撞主要有两种算法（或方法）：边界矩形（也称为边界框）及基于距离的碰撞检测。真正让一个游戏鹤立鸡群的是程序对碰撞的响应能有多好。通过学习我们会发现使用这两种方法进行碰撞检测有多简单，但这却是我们对精灵（及以后要介绍的网格）进行编程让它在真正的碰撞事件中响应的方法。

首先学习边界框碰撞检测。这种碰撞检测类型的关键在于识别两个精灵在屏幕上的位置，然后比较它们的边界框（或矩形），看是否重叠。这就是这种类型的碰撞检测称为边界框碰撞检测的原因，因为每个精灵都被当成一个逻辑上的盒子，或者说矩形。

如果知道两个精灵的位置并且知道每个精灵的宽度和高度，那么就可以确定两个矩形是否相交。边界矩形碰撞检测描述了使用精灵的边界来进行碰撞检测的方法。要取得精灵的左上角位置，只需知道它的 X 和 Y 值。要取得右下角的值，对 X 和 Y 分别加上宽度和高度值即可。同时，这些值可以以左、上、右和下来表示。

#### 8.1.1 处理矩形

我们将使用 RECT 来表示每个边界框。RECT 有四个属性：left、top、right 和 bottom。下面是它的一个示例：

```
int x = 10, y = 10;
int width = 64, height = 64;
RECT rect;
rect.left = x;
rect.top = y;
rect.right = x + width;
rect.bottom = y + height;
```

也可用如下这种更简单的格式来创建 RECT：

```
RECT rect = { x, y, x + width, y + height };
```

这条语句的结果是生成有如下这些值的 RECT：

```
left = 10  
top = 10  
right = 74  
bottom = 74
```

在创建一个表示精灵边界框的矩形时，要将这个矩形的左和上属性设置为精灵的 x 和 y 值，而右下角则设置为精灵的位置加上尺寸值。结果就是一个在逻辑上包围了屏幕上的精灵的矩形。

要实际应用这些代码，需要调用 Windows API 函数。这个函数极为有用，因为它只需一个调用就为我们执行了碰撞测试！这个函数名称为 `IntersectRect`。它接受两个 RECT 变量并且简单地返回 0 和 1 表示假和真（表示精灵有碰撞）。这个函数也返回两个精灵的交集，也就是重叠的部分，但我们对这一信息不感兴趣（简单的是和不是就足够了！）。

不过，我们一直使用全局变量来处理精灵——纹理、位置、尺寸、旋转和缩放等。我们不太可能将所有这些变量传递给一个碰撞测试函数。所以，需要有个精灵结构来包含所有这些属性。这个结构就如第 5 章中所用的 BOMB 结构那样：

```
struct BOMB  
{  
    float x,y;  
  
    void reset()  
    {  
        x = (float)(rand() % (SCREENW-128));  
        y = 0;  
    }  
};
```

我很喜欢这个改进过的程序，有了它可以很容易地保存炸弹的位置及使用 `reset()` 函数随机地移动它。我们需要有一个相似的结构用于更为一般化的精灵，然后在需要时给它增加新功能。那么，对于初学者来说，我们要编写这个基础结构的代码，让它先具备测试碰撞所需的精灵属性：

```
struct SPRITE  
{  
    float x,y;  
    int width, height;  
};
```

### 8.1.2 编写碰撞函数

现在来看一个在这些精灵属性的基础上创建两个矩形并且调用 `IntersectRect` 来检查它们是否碰撞的函数。这个函数名为 `Collision`，它的可重用性非常好（即使更改 `SPRITE` 结构）：

```

int Collision( Sprite sprite1, Sprite sprite2)
{
    RECT rect1;
    rect1.left = sprite1.x;
    rect1.top = sprite1.y;
    rect1.right = sprite1.x + sprite1.width;
    rect1.bottom = sprite1.y + sprite1.height;

    RECT rect2;
    rect2.left = sprite2.x;
    rect2.top = sprite2.y;
    rect2.right = sprite2.x + sprite2.width;
    rect2.bottom = sprite2.y + sprite2.height;

    RECT dest; //ignored
    return IntersectRect(&dest, &rect1, &rect2);
}

```

**建议** 你将注意到在 Collision 函数上有编译器警告信息，因为 Sprite.x 和 Sprite.y 属性是浮点数，而 RECT 属性是长整型。要想去除这些警告，可将精灵属性类型强制转换成 long。

### 8.1.3 新的精灵结构

为了至少能够处理在前面的章节中给出的所有功能，我们需要对 Sprite 结构添加新的特性。随着需求的增加，我们可以在其最初的定义后面添加新内容。对于这个结构，我喜欢它的地方在于，一个简单的构造函数就设置好了一个简单的精灵（不带任何动画）所需的所有初始条件。因为无需每次手动设置所有的属性，所以使用这个结构来处理简单的精灵会很容易。

```

struct Sprite
{
    float x,y;
    int frame, columns;
    int width, height;
    float scaling, rotation;
    int startframe, endframe;
    int starttime, delay;
    int direction;
    float velx, vely;
    D3DCOLOR color;

    Sprite()
    {
        frame = 0;
        columns = 1;
        width = height = 0;
        scaling = 1.0f;
        rotation = 0.0f;
        startframe = endframe = 0;
        direction = 1;
        starttime = delay = 0;
        velx = vely = 0.0f;
        color = D3DCOLOR_XRGB(255,255,255);
    }
};

```



### 8.1.4 为精灵的缩放进行调整

如果更改精灵的比例，那么 Collision 函数就无法正确工作，因为它只认识原来的边界框。需要调整 Collision 函数以便它能计算缩放因子。从技术上说，应该在进行了任何类型的变换之后都重新计算边界框，因为（尤其是）旋转会更改边界框的尺寸，但这里我们将忽略旋转。我们真正关心的只是缩放，因为它对碰撞的有效性绝对有影响。

为了计入缩放因子，我们必须在创建边界框（以 RECT 的形式）时将精灵的宽度和高度乘以缩放值。为了让使用默认比例的精灵正常工作，我们必须确认将默认缩放值设置为 1.0（你可能会错误地设置成 0，因为将变量初始化为 0 很常见。但缩放因子必须是 1，它表示其尺寸的 100%）。如果创建并测试碰撞的是简单的精灵，那么默认的 1.0 就可以了。但如果创建的精灵使用了其他因子值来缩放（比 1.0 大或小），那么就应该计算 Collision 函数中的缩放因子。以下是新的版本，更改用粗体表示：

```
int Collision(SPRITE sprite1, SPRITE sprite2)
{
    RECT rect1;
    rect1.left = (long)sprite1.x;
    rect1.top = (long)sprite1.y;
    rect1.right = (long)sprite1.x + sprite1.width * sprite1.scaling;
    rect1.bottom = (long)sprite1.y + sprite1.height * sprite1.scaling;

    RECT rect2;
    rect2.left = (long)sprite2.x;
    rect2.top = (long)sprite2.y;
    rect2.right = (long)sprite2.x + sprite2.width * sprite2.scaling;
    rect2.bottom = (long)sprite2.y + sprite2.height * sprite2.scaling;

    RECT dest; //ignored
    return IntersectRect(&dest, &rect1, &rect2);
}
```

### 8.1.5 边界框演示程序

我们需要用一个完整（但简单）的示例来演示这种碰撞检测类型。这里有一个名为 Bound Box Demo 的程序，其源代码列在下面。这个程序需要对框架文件做一点更改以便将 Collision 函数集成到我们的游戏库中。可使用 CD-ROM 中的 \chapter08 文件夹中的 DirectX\_Project 模板作为本项目的开始点（为了方便使用，它已经更新到第 7 章的内容）。

#### 1. 在 MyDirectX.h 中添加代码

打开 MyDirectX.h 头文件并添加以下函数原型：

```
//bounding box collision detection
int Collision(SPRITE sprite1, SPRITE sprite2);
```

也将 SPRITE 结构添加到这个文件中，这样在主程序中就可以见到它，而且对 Collision 函数可用。可以在 MyDirectX.h 文件中靠近顶部的任何地方添加 SPRITE 结构：

```
//sprite structure
struct SPRITE
{
    float x,y;
    int frame, columns;
    int width, height;
    float scaling, rotation;
    int startframe, endframe;
    int starttime, delay;
    int direction;
    float velx, vely;
    D3DCOLOR color;

    SPRITE()
    {
        frame = 0;
        columns = 1;
        width = height = 0;
        scaling = 1.0f;
        rotation = 0.0f;
        startframe = endframe = 0;
        direction = 1;
        starttime = delay = 0;
        velx = vely = 0.0f;
        color = D3DCOLOR_XRGB(255,255,255);
    }
};
```

## 2. 在 MyDirectX.cpp 中添加代码

接下来，打开 MyDirectX.cpp 文件并添加完整的函数：

```
//bounding box collision detection
int Collision(SPRITE sprite1, SPRITE sprite2)
{
    RECT rect1;
    rect1.left = (long)sprite1.x;
    rect1.top = (long)sprite1.y;
    rect1.right = (long)sprite1.x + sprite1.width * sprite1.scaling;
    rect1.bottom = (long)sprite1.y + sprite1.height * sprite1.scaling;

    RECT rect2;
    rect2.left = (long)sprite2.x;
    rect2.top = (long)sprite2.y;
    rect2.right = (long)sprite2.x + sprite2.width * sprite2.scaling;
    rect2.bottom = (long)sprite2.y + sprite2.height * sprite2.scaling;

    RECT dest; //ignored
    return IntersectRect(&dest, &rect1, &rect2);
}
```

## 3. MyGame.cpp

现在我们可以把重点放在边界框演示程序的 MyGame.cpp 文件中的源代码上了。这个程序做



的是一艘用户可在屏幕上上下下移动的飞船及两颗从左到右移动的星星。在把飞船移到星星的路径上时，这两个精灵将会碰撞并且导致星星以相反的方向弹回。图 8-1 显示了运行中的程序。

```
#include "MyDirectX.h"
using namespace std;

const string APPTITLE = "Bounding Box Demo";
const int SCREENW = 1024;
const int SCREENH = 768;

SPRITE ship, asteroid1, asteroid2;
LPDIRECT3DTEXTURE9 imgShip = NULL;
LPDIRECT3DTEXTURE9 imgAsteroid = NULL;

bool Game_Init(HWND window)
{
    //initialize Direct3D
    Direct3D_Init(window, SCREENW, SCREENH, false);

    //initialize DirectInput
    DirectInput_Init(window);

    //load the sprite textures
    imgShip = LoadTexture("fatship.tga");
    if (!imgShip) return false;
    imgAsteroid = LoadTexture("asteroid.tga");
    if (!imgAsteroid) return false;

    //set properties for sprites
    ship.x = 450;
    ship.y = 300;
    ship.width = ship.height = 128;
    asteroid1.x = 50;
    asteroid1.y = 200;
    asteroid1.width = asteroid1.height = 60;
    asteroid1.columns = 8;
    asteroid1.startframe = 0;
    asteroid1.endframe = 63;
    asteroid1.velx = -2.0f;

    asteroid2.x = 900;
    asteroid2.y = 500;
    asteroid2.width = asteroid2.height = 60;
    asteroid2.columns = 8;
    asteroid2.startframe = 0;
    asteroid2.endframe = 63;
    asteroid2.velx = 2.0f;

    return true;
}

void Game_Run(HWND window)
{
    if (!d3ddev) return;
```



```

DirectInput_Update();
d3ddev->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
    D3DCOLOR_XRGB(0,0,100), 1.0f, 0);

//move the ship up/down with arrow keys
if (Key_Down(DIK_UP))
{
    ship.y -= 1.0f;
    if (ship.y < 0) ship.y = 0;
}

if (Key_Down(DIK_DOWN))
{
    ship.y += 1.0f;
    if (ship.y > SCREENH - ship.height)
        ship.y = SCREENH - ship.height;
}

//move and animate the asteroids
asteroid1.x += asteroid1.velx;
if (asteroid1.x < 0 || asteroid1.x > SCREENW-asteroid1.width)
    asteroid1.velx *= -1;
Sprite_Animate(asteroid1.frame, asteroid1.startframe, asteroid1.endframe,
    asteroid1.direction, asteroid1.starttime, asteroid1.delay);

asteroid2.x += asteroid2.velx;
if (asteroid2.x < 0 || asteroid2.x > SCREENW-asteroid2.width)
    asteroid2.velx *= -1;
Sprite_Animate(asteroid2.frame, asteroid2.startframe, asteroid2.endframe,
    asteroid2.direction, asteroid2.starttime, asteroid2.delay);

//test for collisions
if (Collision(ship, asteroid1))
    asteroid1.velx *= -1;

if (Collision(ship, asteroid2))
    asteroid2.velx *= -1;

if (d3ddev->BeginScene())
{
    spriteobj->Begin(D3DXSPRITE_ALPHABLEND);

    Sprite_Transform_Draw(imgShip, ship.x, ship.y, ship.width,
        ship.height, ship.frame, ship.columns);

    Sprite_Transform_Draw(imgAsteroid, asteroid1.x, asteroid1.y,
        asteroid1.width, asteroid1.height, asteroid1.frame,
        asteroid1.columns);

    Sprite_Transform_Draw(imgAsteroid, asteroid2.x, asteroid2.y,
        asteroid2.width, asteroid2.height, asteroid2.frame,
        asteroid2.columns);

    spriteobj->End();
    d3ddev->EndScene();
}

```

```
        d3ddev->Present(NULL, NULL, NULL, NULL);
    }

    if (KEY_DOWN(VK_ESCAPE)) gameover = true;
    if (controllers[0].wButtons & XINPUT_GAMEPAD_BACK)
        gameover = true;
}

void Game_End()
{
    if (imgShip) imgShip->Release();
    if (imgAsteroid) imgAsteroid->Release();

    DirectInput_Shutdown();
    Direct3D_Shutdown();
}
```

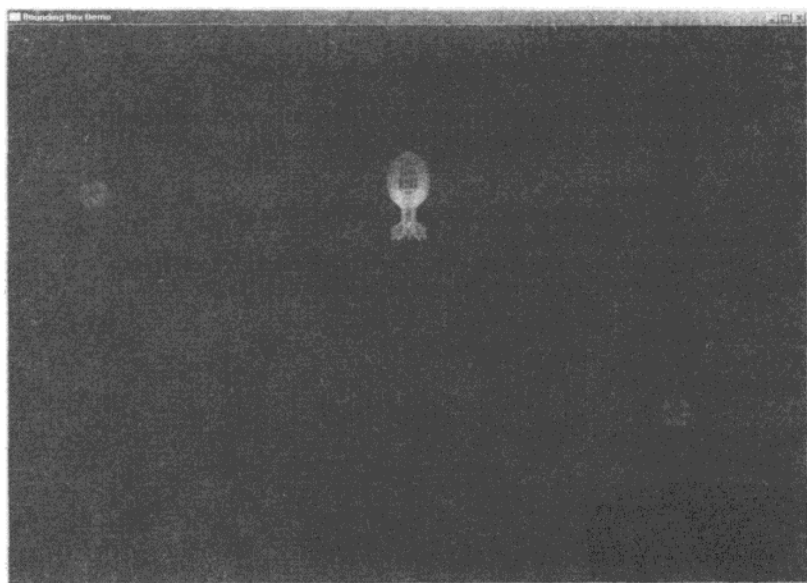


图 8-1 Bounding Box Demo 程序演示基本的碰撞检测

## 8.2 基于距离的碰撞检测

边界框碰撞检测产生相当精确的碰撞结果，而且非常快速。但有些情况下这种方法不能很好地适应，例如使用带有圆角的美工作品或非常复杂的形状（例如有突出机翼的飞机）。在非常情况下，还需要有另外的碰撞检测方式，此时我们可以使用基于距离的碰撞算法。

在使用距离确定两个精灵是否碰撞时，我们必须计算每个精灵的中心点，计算精灵的半径（从中心点到边缘），然后检查两个中心点之间的距离。如果这个距离少于两个半径的和，那么

就可以肯定两个精灵有重叠。为什么呢？因为每个精灵的半径加起来必须小于两个精灵之间的距离。

### 8.2.1 计算距离

要计算任意两点之间的距离，只需参考经典的数学距离公式即可。通过将两点作为直角三角形两条边的顶点，任意两点都可转换为直角三角形。取得每个点的 X 和 Y 的 delta 值（差值），将每个 delta 值平方然后相加，然后求其平方根，就是两点间的距离。见图 8-2。

```
delta_x = x1 - x2
delta_y = y1 - y2
distance = square root ( (delta_x * delta_x) + (delta_y * delta_y) )
```

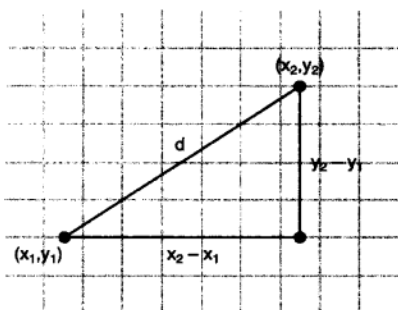


图 8-2 通过形成一个连接它们的直角三角形计算两点间距离

### 8.2.2 编写计算距离的代码

我们可以把这些内容编写到函数中，它使用两个精灵作为参数并从精灵的属性计算出 delta 值和距离。也必须将缩放因子考虑在内，就如在边界框碰撞检测中所做的一样。此外，精灵的最大尺寸（不是宽度就是高度）将用于计算半径。首先计算第一个精灵的半径：

```
if (sprite1.width > sprite1.height)
    radius1 = (sprite1.width * sprite1.scaling) / 2.0;
else
    radius1 = (sprite1.height * sprite1.scaling) / 2.0;
```

有了半径之后，再计算第一个精灵的中心点。中心值储存在向量中：

```
double x1 = sprite1.x + radius1;
double y1 = sprite1.y + radius1;
D3DXVECTOR2 vector1(x1, y1);
```

这个名为 vector1 的向量包含了第一个精灵的中心点，无论它是否位于屏幕上。将相同的代码复制给第二个精灵，我们就会得到两个精灵的中心点和半径。有了这些值，就可以开始进行距离计算了。首先计算 X 和 Y 的 delta 值：

```
double deltax = vector1.x - vector2.x;
double deltax = vector2.y - vector1.y;
```

有了这些 delta 值计算距离就非常容易了：

```
double dist = sqrt((deltax * deltax) + (deltay * deltax));
```

我们将此编写到一个可重用的函数中。这个函数命名为 CollisionD 以便与边界框的版本（名为 Collision）区分开来。你可能更常使用 Collision 函数，因为它毕竟要快得多。

```
bool CollisionD(SPRITE sprite1, SPRITE sprite2)
{
    double radius1, radius2;

    //calculate radius 1
    if (sprite1.width > sprite1.height)
        radius1 = (sprite1.width * sprite1.scaling) / 2.0;
    else
        radius1 = (sprite1.height * sprite1.scaling) / 2.0;

    //center point 1
    double x1 = sprite1.x + radius1;
    double y1 = sprite1.y + radius1;
    D3DXVECTOR2 vector1(x1, y1);

    //calculate radius 2
    if (sprite2.width > sprite2.height)
        radius2 = (sprite2.width * sprite2.scaling) / 2.0;
    else
        radius2 = (sprite2.height * sprite2.scaling) / 2.0;

    //center point 2
    double x2 = sprite2.x + radius2;
    double y2 = sprite2.y + radius2;
    D3DXVECTOR2 vector2(x2, y2);

    //calculate distance
    double deltax = vector1.x - vector2.x;
    double deltax = vector2.y - vector1.y;
    double dist = sqrt((deltax * deltax) + (deltay * deltax));

    //return distance comparison
    return (dist < radius1 + radius2);
}
```

现在将这个函数复制到框架中。将这个函数添加到 MyDirectX.cpp 中，将原型添加到 MyDirectX.h 中：

```
bool CollisionD(SPRITE sprite1, SPRITE sprite2);
```

### 8.2.3 测试基于距离的碰撞

可以用一个新的示例程序演示基于距离的碰撞检测，不过它是基于前面的边界框碰撞演示

的，而且只涉及两行代码的更改。所以不再在这里重复这个程序的代码清单。如果从 CD-ROM 中装载 Distance\_Demo 项目，只需留意如下两行代码的不同：它们调用新的 CollisionD 函数。否则，这两个程序一模一样：

```
//test for collisions
if (CollisionD(ship, asteroid1))
    asteroid1.velx *= -1;

if (CollisionD(ship, asteroid2))
    asteroid2.velx *= -1;
```

### 8.3 你所学到的

本章我们学习了两种常见的 2D 精灵的碰撞检测形式：边界框和距离。虽然本章中的两个示例程序演示了实现碰撞的方法，但我们还没在真实的游戏见到它们，所以，要在目前这个早期阶段将碰撞检测（和响应）的概念在游戏项目中的作用说明白有点困难。先把它放一边，我们将在第 16 章讨论一个完整的游戏。

本章有以下几个要点：

- 基于边界框的碰撞检测
- 基于距离的碰撞检测

### 8.4 复习测验

以下这些复习测验题可用于检验读者从本章学到了哪些东西。

- 1) 在使用 IntersectRect 函数时，为每个精灵填充边界值的对象是什么类型？
- 2) 传递给 IntersectRect 的第一个参数是什么？
- 3) 计算两点之间的距离所用的三角形是什么类型的（概念上的）？
- 4) 简要描述边界框方式处理精灵缩放的方法。
- 5) 在快节奏的、每次在屏幕上有上百个精灵的街机游戏中，精度并不是那么重要，那么应该使用两种碰撞检测方法中的哪一种？
- 6) 在节奏慢一点的游戏（例如 RPG 游戏），玩游戏时精度很重要，而且在屏幕上一个时间内显示的精灵不多，那么应该使用两种碰撞检测方法中的哪一种？
- 7) 在计算两个精灵之间的距离时，每个精灵上的（X，Y）点通常位于何处？
- 8) 在两个精灵发生碰撞之后，在下一帧之前为什么要将精灵彼此移开？
- 9) IntersectRect 函数的第二个和第三个参数是什么？
- 10) 简要描述游戏中需要使用两种碰撞检测技术来检测相同的两个精灵是否碰触的情况。

### 8.5 自己动手

以下练习可考验你对本章中的内容的掌握程度。

习题 1 Bounding Box Demo 项目是个非常简单的项目，只为了正确演示碰撞检测的工作方

式。我们可以在它上面做一些更有趣的事情。例如，修改程序，让星星在屏幕上以随机的方向运动而不仅仅是在固定的行上水平移动。

习题 2 Distance Demo 使用相同的项目代码演示了基于距离的碰撞检测。我们来使它物尽其用。修改程序，让它在调用距离函数时使每个精灵使用更小的半径值。可尝试一半或者四分之一的原半径值。看看会有什么发生？



## 第9章 打印文本

本章我们将学习使用 ID3DXFont 创建字体并在屏幕上打印文本的方法。这个类使我们可能使用任何安装在 Windows 系统中的 TrueType 字体来打印文本，不过建议只使用标准字体（例如 Times New Roman 和 Verdana），这样文本在每台 PC 上都能按我们预想的样子显示（如果使用不常用的字体而系统中没有该字体的话，那么 Windows 会尝试使用接近它的字体，这不是我们希望在游戏中发生的）。

本章将学到：

- 如何使用 ID3DXFont 创建新字体。
- 如何使用字体将文本打印在屏幕上。
- 如何让文本在某个区域内折行。

### 9.1 创建字体

我们将通过 ID3DXFont 接口使用任何想用的、安装在系统中的 TrueType 字体将文本打印到屏幕上。在过去我喜欢使用基于位图的字体，其字体集以 ASCII 顺序储存在一个位图文件中。在装载了这个位图之后，这种字体借用精灵的特性将字符串中的每个字符渲染出来。这样做的结果是我们得到一种在显示效果上和我们所期望的完全一致的字体，因为是在控制着源位图。

**建议** ASCII 表示美国标准信息交换码（American Standard Code for Information Interchange），它是字符集编码的标准。例如，空格的 ASCII 码是 32，回车的 ASCII 码是 13。ASCII 码与 Windows 虚拟键盘码及 DirectInput 键盘码不同（它们是 Windows 专有的）。

DirectX 提供了一个字体类，它抽象了整个过程，从而使我们可以不必太关注其内部逻辑（例如满载字体的位图图像）而花更多的时间在游戏代码上。ID3DXFont 接口用于创建字体，其指针版已经预定义好了：

```
LPD3DXFONT font;
```

我们将使用一个名为 D3DXCreateFontIndirect 的函数来创建字体并且为字体打印做准备。不过在做这件事之前，我们必须先使用 D3DXFONT\_DESC 结构来设置想要的字体属性。

#### 9.1.1 字体描述符

D3DXFONT\_DESC 结构由以下属性组成：

- INT Height
- UINT Width
- UINT Weight



- UINT MipLevels
- BOOL Italic
- BYTE CharSet
- BYTE OutputPrecision
- BYTE Quality
- BYTE PitchAndFamily
- CHAR FaceName[LF\_FACESIZE]

别被这些描述符属性给吓着了，因为它们当中的大多数都设为零或者默认值。只有两个属性是真正重要的：Height 和 FaceName。以下是字体描述符变量的一个示例，它被初始化成 Arial 24 点字体的值。

```
D3DXFONT_DESC desc = {
    24,                //height
    0,                 //width
    0,                 //weight
    0,                 //miplevels
    false,             //italic
    DEFAULT_CHARSET,   //charset
    OUT_TT_PRECIS,     //output precision
    CLIP_DEFAULT_PRECIS, //quality
    DEFAULT_PITCH,     //pitch and family
    "Arial"            //font name
};
```

### 9.1.2 创建字体对象

在设置了字体描述符之后，就可以使用 D3DXCreateFontIndirect 函数来创建字体对象了。这个函数需要三个参数：

- Direct3D 设备
- D3DXFONT\_DESC
- LPD3DXFONT

我们来看一个使用这个函数创建字体的示例：

```
D3DXCreateFontIndirect(d3ddev, &desc, &font);
```

### 9.1.3 可重用的 MakeFont 函数

将所有这些代码放到一个可重用的函数中，然后添加到游戏库中。这个函数需要字体名称及字体点尺寸作为参数，它返回一个指向 LPD3DXFONT 对象的指针。

```
LPD3DXFONT MakeFont(string name, int size)
{
    LPD3DXFONT font = NULL;
    D3DXFONT_DESC desc = {
        size,                //height
        0,                   //width
```

```

0,                //weight
0,                //miplevels
false,            //italic
DEFAULT_CHARSET,  //charset
OUT_TT_PRECIS,    //output precision
CLIP_DEFAULT_PRECIS, //quality
DEFAULT_PITCH,    //pitch and family
**               //font name
};
strcpy(desc.FaceName, name.c_str());
D3DXCreateFontIndirect(d3ddev, &desc, &font);
return font;
}

```

## 9.2 使用 ID3DXFont 打印文本

我们已经学习了创建字体对象的方法，但还不知道如何用它将文本打印到屏幕上。本节将学习如何使用一个已有字体来打印文本（这个字体先前创建并初始化过）。可以使用 ID3DXFont::DrawText() 函数。DrawText 函数需要下面这些属性：

LPD3DXSPRITE pSprite	精灵渲染对象
LPCSTR pString	要打印的文本
INT count	文本长度
LPRECT pRect	指定位置和边界的矩形
DWORD format	诸如 DT_WORDBREAK 这样的格式化选项
D3DCOLOR color	文本的输出颜色

### 9.2.1 使用 DrawText 打印

假设已经创建了名为 font 的字体对象，并且想使用 DrawText 函数打印一些东西。以下是示例：

```

RECT rect = { 10, 10, 0, 0 };
D3DCOLOR white = D3DCOLOR_XRGB(255,255,255);
string text = "This is a text message that will be printed.";
font->DrawText(
    spriteobj,
    text.c_str(),
    text.length(),
    &rect,
    DT_LEFT,
    white
);

```

可以将这些代码包装成一个可重用函数。以下是示例代码，最后一个参数指定颜色，如果忽略默认是白色。

```

void Print(
    LPD3DXFONT font,
    int x,
    int y,
    string text,
    D3DCOLOR color = D3DCOLOR_XRGB(255,255,255))
{
    //figure out the text boundary
    RECT rect = { x, y, 0, 0 };
    font->DrawText(NULL, text.c_str(), text.length(), &rect, DT_CALCRECT, color);

    //print the text
    font->DrawText(spriteobj, text.c_str(), text.length(), &rect, DT_LEFT,
        color);
}

```

### 9.2.2 文本折行

如果想让文本在一个定义了文本边界的矩形区域内格式化，有一个选项可以利用。如果设置了它，那么文本将自动在每个词（以空格作为分隔符）之后折行<sup>⊖</sup>。举个例子来说，在使用我们自定义的 GUI 控件时这会非常有用。通常，这个矩形被定义为宽度和高度为零，也就是根本不使用边界。但如果指定了宽度和高度，并且使用 DT\_CALCRECT 选项，那么 DrawText 将自动进行词折行！以下是示例：

```

RECT rect = { 60, 250, 350, 700 };
D3DCOLOR white = D3DCOLOR_XRGB(255,255,255);
string text = "This is a long string that will be wrapped.";
font->DrawText(
    spriteobj,
    text.c_str(),
    text.length(),
    &rect,
    DT_WORDBREAK,
    white);

```

注意，RECT 被定义为宽度为 290 像素（left = 60，right = 350），而 bottom 属性被设为一个很高的值（在这里它不像宽度那么重要）。在 DrawText 执行的过程中，如果到达右边缘（在本例中是 350），那么它将找到最近的空格字符并将其后的词折到下一行。DrawText 通过打印整个词并在绘制每个字符之前计算每个词占用空间是否合适来完成这一任务。这是个极为有帮助的功能！

## 9.3 测试字体输出

现在来看一个示例程序，它演示了 ID3DXFont::DrawText 的大多数选项。首先将字体函数添加到助手文件中，以便能在将来的项目中使用它。打开 MyDirectX.h 头文件然后添加函数原型：

⊖ 以便让文本显示在矩形区域内，也就是一种“包裹”的效果。——译者注

```
//font functions
LPD3DXFONT MakeFont(
    string name,
    int size
);
void FontPrint(
    LPD3DXFONT font,
    int x,
    int y,
    string text,
    D3DCOLOR color = D3DCOLOR_XRGB(255,255,255)
);
```

接下来，打开 MyDirectX.cpp 文件并添加如下函数：

```
LPD3DXFONT MakeFont(string name, int size)
{
    LPD3DXFONT font = NULL;
    D3DXFONT_DESC desc = {
        size,                //height
        0,                   //width
        0,                   //weight
        0,                   //miplevels
        false,               //italic
        DEFAULT_CHARSET,     //charset
        OUT_TT_PRECIS,       //output precision
        CLIP_DEFAULT_PRECIS, //quality
        DEFAULT_PITCH,       //pitch and family
        **                   //font name
    };
    strcpy(desc.FaceName, name.c_str());
    D3DXCreateFontIndirect(d3ddev, &desc, &font);
    return font;
}

void FontPrint(LPD3DXFONT font, int x, int y, string text, D3DCOLOR color)
{
    //figure out the text boundary
    RECT rect = { x, y, 0, 0 };
    font->DrawText( NULL, text.c_str(), text.length(), &rect, DT_CALCRECT, color);

    //print the text
    font->DrawText(spriteobj, text.c_str(), text.length(), &rect, DT_LEFT,
        color);
}
```

字体支持函数现在已经位于游戏库文件中了，可以使用示例文件来测试字体输出。字体演示程序生成的输出如图 9-1 所示。

```
#include "MyDirectX.h"
using namespace std;

const string APPTITLE = "Font Demo";
const int SCREENW = 1024;
```

```

const int SCREENH = 768;

//declare some font objects
LPD3DXFONT fontArial24 = NULL;
LPD3DXFONT fontGaramond36 = NULL;
LPD3DXFONT fontTimesNewRoman40 = NULL;
bool Game_Init(HWND window)
{
    Direct3D_Init(window, SCREENW, SCREENH, false);
    DirectInput_Init(window);

    //create some fonts
    fontArial24 = MakeFont("Arial",24);
    fontGaramond36 = MakeFont("Garamond",36);
    fontTimesNewRoman40 = MakeFont("Times New Roman", 40);

    return true;
}

void Game_Run(HWND window)
{
    //make sure the Direct3D device is valid
    if (!d3ddev) return;

    //update input devices
    DirectInput_Update();

    d3ddev->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
        D3DCOLOR_XRGB(0,0,100), 1.0f, 0);
    //start rendering
    if (d3ddev->BeginScene())
    {
        spriteobj->Begin(D3DXSPRITE_ALPHABLEND);

        //demonstrate font output
        FontPrint(fontArial24, 60, 50,
            "This is the Arial 24 font printed with ID3DXFont");

        FontPrint(fontGaramond36, 60, 100,
            "The text can be printed in any color like this magenta!",
            D3DCOLOR_XRGB(255,0,255));

        FontPrint(fontTimesNewRoman40, 60, 150,
            "Or how about bright green instead?",
            D3DCOLOR_XRGB(0,255,0));

        //demonstrate text wrapping inside a rectangular region
        RECT rect = { 60, 250, 350, 700 };
        D3DCOLOR white = D3DCOLOR_XRGB(255,255,255);
        string text = "This is a long string that will be ";
        text += "wrapped inside a rectangle.";
        fontTimesNewRoman40->DrawText( spriteobj, text.c_str(),
            text.length(), &rect, DT_WORDBREAK, white);

        spriteobj->End();
        d3ddev->EndScene();
    }
}

```

```

        d3ddev->Present(NULL, NULL, NULL, NULL);
    }

    if (KEY_DOWN(VK_ESCAPE)) gameover = true;
    if (controllers[0].wButtons & XINPUT_GAMEPAD_BACK)
        gameover = true;
}

void Game_End()
{
    if (fontAria124) fontAria124->Release();
    if (fontGaramond36) fontGaramond36->Release();
    if (fontTimesNewRoman40) fontTimesNewRoman40->Release();

    DirectInput_Shutdown();
    Direct3D_Shutdown();
}

```

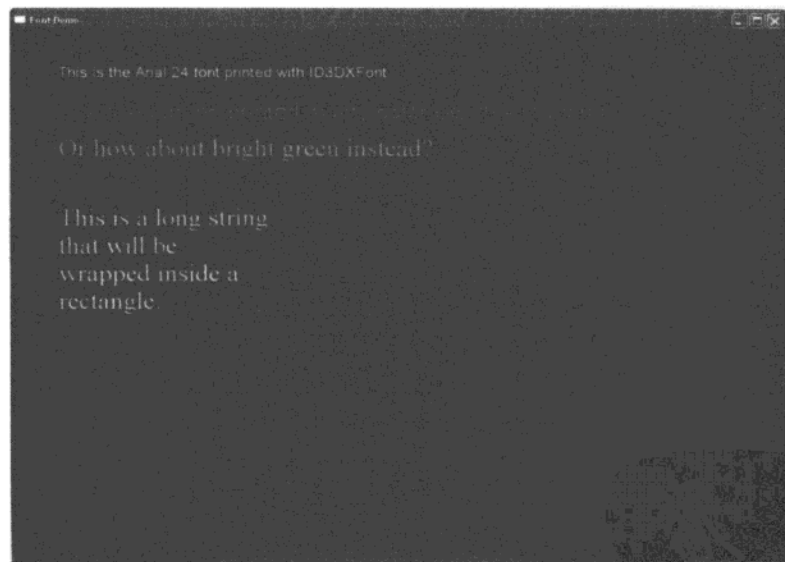


图 9-1 Font Demo 程序演示使用 ID3DXFont 打印文本的方法

## 9.4 你所学到的

本章我们学习了使用 ID3DXFont 在支持 TrueType 字体的基于 Direct3D 的屏幕上打印文本的方法。有以下几个要点：

- 如何创建新字体。
- 如何使用字体打印文本。
- 如何在矩形内实现文本折行。

## 9.5 复习测验

以下这些复习测验题可用于检验读者从本章学到了哪些东西。

- 1) 用于将文本打印在屏幕上的字体对象的名称是什么?
- 2) 字体对象的长指针版本的名称是什么?
- 3) 用于将文本打印在屏幕上的函数名称是什么?
- 4) 用于创建基于特定字体属性的新字体对象的函数是哪个?
- 5) 用于指定文本在屏幕上给定矩形区域中折行的常量名称是什么?
- 6) 如果不给字体渲染器提供精灵对象, 那么在将字体渲染到屏幕上时, 它是否会创建自己的精灵对象用于 2D 输出?
- 7) `std::string` 中哪个函数将字符串数据转换为 C 样式的字符数组, 以便诸如 `strcpy` 这样的函数使用?
- 8) `std::string` 中哪个函数返回字符串的长度 (例如, 字符串中的字符数量)?
- 9) 用于定义文本输出颜色的 Direct3D 数据类型是什么?
- 10) 哪个函数返回带有 alpha 通道成分的 Direct3D 颜色?

## 9.6 自己动手

以下练习可考查你对本章中的内容的掌握程度。

习题 1 修改 Font Test 项目, 让它打印出你的姓名, 而不是演示中提供的示例文本。

习题 2 修改 Font Test 项目, 让它能够如同移动精灵那样在屏幕上移动文本消息。



## 第10章 卷动背景

我们在大多数动作和街机游戏中看到的背景移动效果使用的是基于图片单元的卷动。虽然这种技术已经出现有几十年了，但它仍旧被用来渲染背景，而这类2D游戏当今仍旧频繁活跃着。回到那些老时光，当时计算机内存极为有限，使用基于图片单元的卷动方式是因为它极为高效。今天我们想当然地认为内存应该有数个GB，但那么多的内存在视频游戏的早期岁月里是难以置信的，即使是硬盘也没那么大，更不用说主存储器（RAM）了。我们将在本章学习的**虚拟屏幕缓冲区**的概念，在当时只在非常有限的视频卡中使用（带有256~1024 KB视频内存）。在那时，能有两个320×240的屏幕（或缓冲区）是非常幸运的事，更不用想能有足够的内存来进行大的卷动效果了。本章主要介绍创建卷动背景的两种不同方法：基于位图的和基于图片单元的。

本章将学到：

- 卷动的介绍。
- 创建基于图片单元的背景。
- 使用单一的大卷动缓冲区。
- 使用动态绘制的图片单元。

### 10.1 卷动

卷动是什么？在今天的游戏世界中，3D是每个人的关注点，从来没听说过卷动的游戏者和程序员大有人在。实在是很遗憾啊！就算不被理解或欣赏，但现代游戏又长又迷人的历史传承，在今天仍旧有价值。控制台游戏业为卷动效果倾尽全力并带来极大价值，尤其是在手持系统中，例如Game Boy Advance。就拿不同寻常的GBA销售市场来说，你如果知道在一天中销售的2D游戏数量要超过3D游戏，是否感到惊奇？图10-1显示了卷动的概念。

游戏世界 / 内存位图

1600×1200

(X = 930, Y=520)

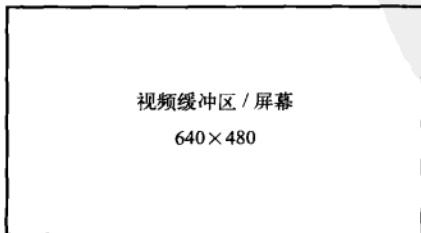


图 10-1 卷动窗口显示大游戏世界中的一部分



**建议** 卷动是一种在屏幕窗口中显示大的虚拟游戏世界的一小部分，然后通过移动窗口中的视图来表现游戏世界内位置变化的过程。

我们可以在虚拟游戏世界中显示一个巨大的位图图像来表示游戏的当前这一关，然后将该虚拟世界的一部分复制（位传输）到屏幕上。这是最简单的卷动形式。另外一种方法是使用图片单元来创建游戏世界，很快就会讲解这种方法。首先，编写一个小程序来演示使用位图卷动的方法。

我们已经看到了简单的卷动器的样子，虽然它需要依靠键盘输入来卷动。高速卷动的街机游戏应该自动地水平或垂直卷动，显示位于游戏者（通常由飞机或飞船来表示）之下的基于地面的、空中的或空间的地形。这些游戏的要点是保持快速动作，让游戏者没有机会从一波又一波的敌人中缓过劲来。随后的两章将主要讨论这些主题！目前，我们暂时将这些问题简化，讲解卷动的基础知识，为以后深入探究这些高级主题做准备。

### 10.1.1 背景和布景

背景由某种形式的影像或地形组成，精灵就绘制在背景之上。背景可以仅仅是游戏动作背后的一幅漂亮的图画，也可以如卷动器那样参与动作。关于卷动器，它并不只是高速街机游戏的专利。角色扮演游戏通常也使用卷动器，还有大多数的体育游戏也是。

背景的设计应围绕游戏的目标来进行。我们不应先找一张好看的背景图然后在此之上构建游戏（不过，我承认游戏经常是如此开始的）。我们不能仅依赖于单一的酷技术来作为整个游戏的基础，否则人们会永远记得这是个时髦的游戏，想靠最新时尚赚钱。我们要设好自己的范例，做出自己的标准；而不是跟随和模仿。

你会问我在说什么呢？你可能会有这样的印象：卷动游戏可以实现的所有一切都已经做了不下十次了。不是的，不是的！还记得 Doom 第一次出现的时候吗？那时所有人都在模仿 Wolfenstein 3D；Carmack 和 Romero 突然跳得高出标杆好几百点并吊足了所有人的胃口，这给游戏业带来了很大冲击，不论是控制台游戏还是 PC 游戏。

你是否真的觉得该做的都已经做了，已经没有革新的空间了，游戏业已经饱和，不可能再作出成功的“独立”游戏了？这可没有阻挡 Bungie 在其第一个游戏项目上的努力。Halo 在游戏历史上留下了一笔，它提高了所有人对更高级的物理和智能对手的期待。如今，在多年之后，还有那种游戏会出现吗？业内最流行的是什词？物理。要是设计的游戏中没有它，这个游戏突然就显得像 20 世纪 90 年代出品的游戏。现在都在拼物理和人工智能，而这些都是从 Halo 开始的。而对于 Halo 来讲这是完美的，我个人还真想不起在 Halo 之前有哪个游戏能有它那种水平的交互。所以，认为我们不能引领游戏的下一次革新或变革，是绝对没有理由的，即使是 2D 游戏。

### 10.1.2 从图片单元创建背景

卷动背景的真实力量来自名为铺砌（tiling）的技术。铺砌是这么一个过程，其中没有真正的背景，而是由图片单元阵列形成要显示的背景。换言之，这是个虚拟的虚拟背景，与完全位图型的背景相比，只需非常少的内存。图 10-2 给出了一个示例。

能否数出构成图 10-2 中背景的图片单元数量？实际上一共有 18 块图片单元构成这一图像。想象一下，整个游戏屏幕只由几个图片单元构成，而结果却很不赖！显然，真正的游戏将不仅只有草、路、河和桥，真正的游戏还会有精灵在背景上移动。来个示例怎么样？我想你会喜欢这个主意。

### 10.1.3 基于图片单元的卷动

Tile\_Static\_Scroll 程序使用图片单元来填充大的背景位图，我们马上就来编写这个程序。它从位图（包含以行和列编排的图片单元）中装载图片单元，然后使用地图数据来填充由内存中的大位图表示的虚拟卷动表面。如图 10-3 所示。

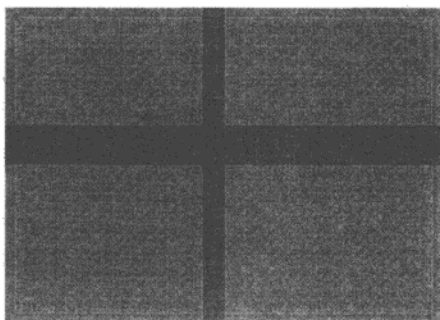


图 10-2 由图片单元构成的位图图像

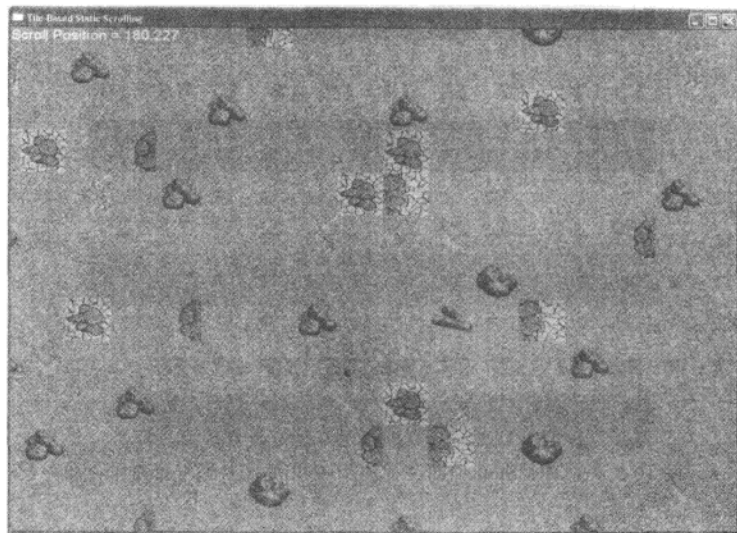


图 10-3 Tile\_Static\_Scroll 程序演示基于图片单元的卷动的方法

这个程序通过将图片单元绘制到创建于内存（实际上是 Direct3D 表面，这里使用表面而不使用纹理是因为这里无需透明度）中的一个大大位图图像来创建图 10-3 所示的由图片单元组成的图片。图 10-4 显示了包含这些图片单元的实际位图。这些图片单元是由 Air Feldman 所创建 (<http://www.flyingyogi.com>)，这是他免费的 SpriteLib 的一部分。

图 10-5 给出了每个图片单元的图例及其值。在构建我们自己的地图时可使用这个图例。

### 10.1.4 基于图片单元的卷动项目

现在，我们编写一个用于演示的测试程序，因为对于想构建一个真正的游戏的人来说，仅有

理论是走不远的。我不了解你的情况，但对于我自己而言，动手要比单纯的阅读学得更好。这里假定你会按照上一章中的同样步骤创建新项目，然后添加需要的库文件。

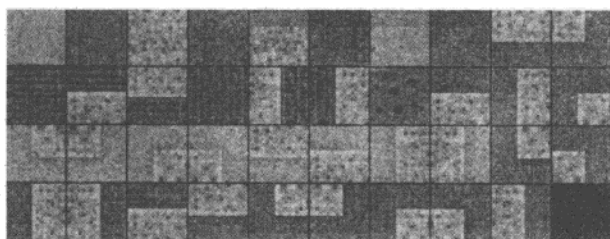


图 10-4 Tile\_Static\_Scroll 程序使用的图片单元的源文件

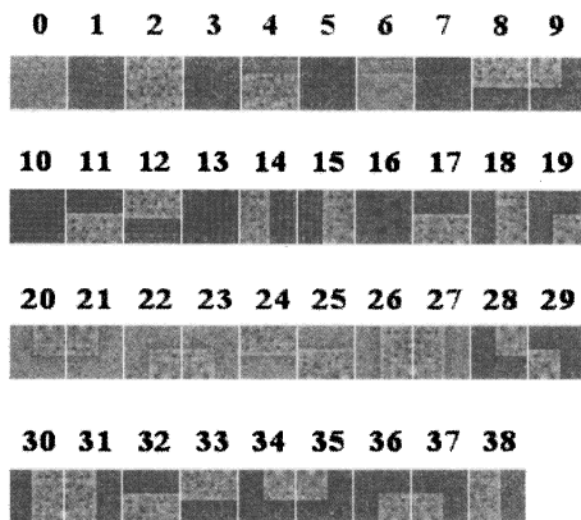


图 10-5 在 Tile\_Static\_Scroll 程序中使用的图片单元地图的图例

```
/*
    Beginning Game Programming, Third Edition
    MyGame.cpp
*/

#include "MyDirectX.h"
#include <sstream>
using namespace std;

const string APPTITLE = "Tile-Based Static Scrolling";
const int SCREENW = 1024;
const int SCREENH = 768;

LPD3DXFONT font;
```



```

        //set destination rect
        RECT r2 = {destx,desty,destx+width,desty+height};

        //draw the tile
        d3ddev->StretchRect(source, &r1, dest, &r2, D3DTEXF_NONE);
    }
void BuildGameWorld()
{
    HRESULT result;
    int x, y;
    LPDIRECT3DSURFACE9 tiles;

    //load the bitmap image containing all the tiles
    tiles = LoadSurface("groundtiles.bmp");

    //create the scrolling game world bitmap
    result = d3ddev->CreateOffscreenPlainSurface(
        GAMEWORLDWIDTH,           //width of the surface
        GAMEWORLDHEIGHT,          //height of the surface
        D3DFMT_X8R8G8B8,
        D3DPPOOL_DEFAULT,
        &gameworld,                //pointer to the surface
        NULL);

    if (result != D3D_OK)
    {
        MessageBox(NULL, "Error creating working surface!", "Error", 0);
        return;
    }

    //fill the gameworld bitmap with tiles
    for (y=0; y < MAPHEIGHT; y++)
        for (x=0; x < MAPWIDTH; x++)
            DrawTile(tiles, MAPDATA[y * MAPWIDTH + x], 64, 64, 16,
                gameworld, x * 64, y * 64);

    //now the tiles bitmap is no longer needed
    tiles->Release();
}

bool Game_Init(HWND window)
{
    Direct3D_Init(window, SCREENW, SCREENH, false);
    DirectInput_Init(window);

    //create pointer to the back buffer
    d3ddev->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, &backbuffer);
    //create a font
    font = MakeFont("Arial", 24);

    BuildGameWorld();

    start = GetTickCount();

    return true;
}

```



```

void Game_End()
{
    if (gameworld) gameworld->Release();
    DirectInput_Shutdown();
    Direct3D_Shutdown();
}

void ScrollScreen()
{
    //update horizontal scrolling position and speed
    ScrollX += SpeedX;
    if (ScrollX < 0)
    {
        ScrollX = 0;
        SpeedX = 0;
    }
    else if (ScrollX > GAMEWORLDWIDTH - SCREENW)
    {
        ScrollX = GAMEWORLDWIDTH - SCREENW;
        SpeedX = 0;
    }

    //update vertical scrolling position and speed
    ScrollY += SpeedY;
    if (ScrollY < 0)
    {
        ScrollY = 0;
        SpeedY = 0;
    }
    else if (ScrollY > GAMEWORLDHEIGHT - SCREENH)
    {
        ScrollY = GAMEWORLDHEIGHT - SCREENH;
        SpeedY = 0;
    }

    //set dimensions of the source image
    RECT r1 = {ScrollX, ScrollY, ScrollX+SCREENW-1, ScrollY+SCREENH-1};

    //set the destination rect
    RECT r2 = {0, 0, SCREENW-1, SCREENH-1};

    //draw the current game world view
    d3ddev->StretchRect(gameworld, &r1, backbuffer, &r2,
        D3DTEXF_NONE);
}

void Game_Run(HWND window)
{
    if (!d3ddev) return;
    DirectInput_Update();
    d3ddev->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
        D3DCOLOR_XRGB(0,0,100), 1.0f, 0);

    //scroll based on key or controller input

```

```

if (Key_Down(DIK_DOWN) || controllers[0].sThumbLY < -2000)
    ScrollY += 1;

if (Key_Down(DIK_UP) || controllers[0].sThumbLY > 2000)
    ScrollY -= 1;

if (Key_Down(DIK_LEFT) || controllers[0].sThumbLX < -2000)
    ScrollX -= 1;

if (Key_Down(DIK_RIGHT) || controllers[0].sThumbLX > 2000)
    ScrollX += 1;

//keep the game running at a steady frame rate
if (GetTickCount() - start >= 30)
{
    //reset timing
    start = GetTickCount();

    //start rendering
    if (d3ddev->BeginScene())
    {
        //update the scrolling view
        ScrollScreen();
        spriteobj->Begin(D3DXSPRITE_ALPHABLEND);

        std::ostringstream oss;
        oss << "Scroll Position = " << ScrollX << ", " << ScrollY;
        FontPrint(font, 0, 0, oss.str());

        spriteobj->End();

        //stop rendering
        d3ddev->EndScene();
        d3ddev->Present(NULL, NULL, NULL, NULL);
    }
}

//to exit
if (KEY_DOWN(VK_ESCAPE) ||
    controllers[0].wButtons & XINPUT_GAMEPAD_BACK)
    gameover = true;
}

```

## 10.2 动态渲染图片单元

只是为了验证理论而显示图片单元不会起实际作用。是的，我们需要有一些创建虚拟背景的代码，将图片单元装载到它上面，然后卷动整个游戏世界。在过去，我曾经使用源代码生成了一幅看起来很真实的游戏地图。我使用了一种算法来匹配地形曲线和直线（例如路、桥和河流），也就是说从头创建了一幅很好的地图，这一切都是我自己做的。构建出算法型的风景是一方面，但在运行时构造出它却不是个好的解决方案，即使地图生成例程做得非常好。

举例来说,在许多游戏中,例如 Warcraft III、Age of Mythology 和 Civilization IV 可飞快地生成游戏世界。显然,程序员花费了大量时间让世界生成例程变得完美。如果游戏能从具备随机生成游戏世界这样的功能中获益,那么这样做是合适的,结果也是值得的。只要有时间进行开发,需要做的只是那些设计上的考虑。

### 10.2.1 图片单元地图

如果还没有生成随机地图的方法(或者就是不想走那条路线),那么可以简单地在数组中创建它,就如我们在上一个项目中所做的那样。但地图数据实际是从哪儿来的呢?而且,更进一步说,我们要从哪儿开始?首先必须意识到,图片单元都是编号的,应该在地图数组中以这种编号方式引用。图片单元地图中的每个数字表示位图文件中的一个图片单元图像。以下给出的是 Tile\_Dynamic\_Scroll 程序(很快会介绍)中定义的数组。

```
int MAPDATA[MAPWIDTH*MAPHEIGHT] = {
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,
33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,
49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,
65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,
81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,
97,98,99,100,101,102,103,104,105,106,107,108,109,110,111,112,
113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,
129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,144,
145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,160,
161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,176,
177,178,179,180,181,182,183,184,185,186,187,188,189,190,191,192,
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,
17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,
33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,
49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,
65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,
81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,
97,98,99,100,101,102,103,104,105,106,107,108,109,110,111,112,
113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,
129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,144,
145,146,147,148,149,150,151,152,153,154,155,156,157,158,159,160,
161,162,163,164,165,166,167,168,169,170,171,172,173,174,175,176,
177,178,179,180,181,182,183,184,185,186,187,188,189,190,191,192
};
```

这里的技巧在于,这只是个一维数组,但其列表方式使地图的样式显得很直观(地图的效果见图 10-6),因为每一行有 16 个数字,这与位图文件中每一行的图片单元数量一致。这么做是为了你在创建自己的地图时可以用它作为模板。如果需要,还可以创建多张地图。只需更改每个地图的名称并且引用想要绘制的地图,新的地图就可绘制出来。这里并没有限制在每一行中不能增加更多图片单元。你还可以尝试将 MAPDATA 做成包含许多地图的二维数组,然后在运行时更换地图!如果你有足够的编写成百上千种不同游戏的创意和魄力,那么这段简单的卷动代码可作为它们的基础。





图 10-6 用于在 Tile\_Dynamic\_Scroll 程序中创建图片单元地图的 Starfield 图像  
(感谢 Space Telescope Science Institute 提供, www.scsti.edu)

## 10.2.2 使用 Mappy 创建图片单元地图

下面将带领读者使用特别棒(而且免费)的图片单元编辑程序 Mappy 按步骤创建一个非常简单的图片单元地图。这个程序可从 <http://www.tilemap.co.uk> 获得, 而且也在 CD-ROM 的 \software\Mappy 中提供了它。这是我最喜欢的基于图片单元游戏的游戏关编辑程序, 也是许多专业游戏开发人员的选择(尤其是开发手持游戏和战略游戏的人员)。在此没有时间给出完整的 Mappy 用法教程, 但它真的是挤满了各种令人惊奇的功能(全都藏在各个子菜单中)。这里只能对 Mappy 做简单介绍, 读入一个大的照片文件然后将其转换为图片单元地图。

**注意** 如果乐于进行基于图片单元的游戏关编辑和开发, 这里推荐《Game Programming All in One》一书第 3 版, 它包含许多关于滚动背景的章节, 有一章全面讲授了 Mappy 的使用方法。

首先启动 Mappy。然后打开 File 菜单并选择 New Map。系统显示如图 10-7 所示的 New Map 对话框。在图中键入 64×64 作为图片单元尺寸、16×24 作为地图尺寸(图片单元地图中图片单元的数量)。然后将会创建新的地图, 但还没有图片单元存在, 如图 10-8 所示。

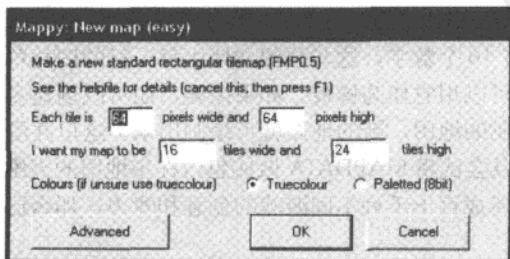


图 10-7 使用 Mappy 创建新地图

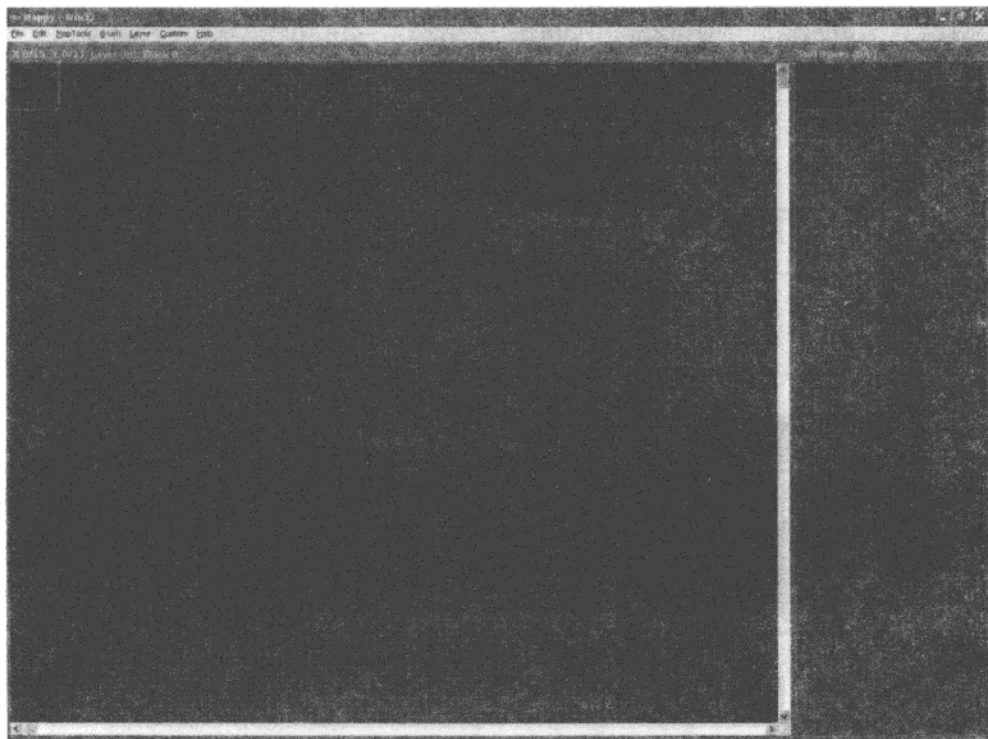


图 10-8 Mappy 已创建新地图，正在等待用户的图片单元

### 1. 导入现有的位图文件

接下来，把一张由哈勃望远镜拍摄的太空图片导入 Mappy，然后将它转换为图片单元地图。如图 10-9 所示，打开 MapTools 菜单，选择 Useful Functions，然后选择 Create map from big picture 选项。浏览位于 CD-ROM 的 \sources\chapter10\Tile Dynamic Scrolling Demo\map 上的 space1.bmp 文件。在选择这个文件时，Mappy 将把它导入到图片单元画板中，如图 10-10 所示。

如图 10-10 所示，有许多图片单元组成了这幅图像！如果对画板中图片单元的数量感到好奇的话，我们就来看看！打开 MapTools 菜单并选择 Map Properties。系统显示 Map Properties 对话框，如图 10-11 所示。看对话框左边的文本值：Map Array、Block Str、Graphics 等。Map Array 文本告诉我们地图的尺寸（正是我们指定的  $16 \times 24$ ，以图片单元为单位）。现在看一看 Graphics 信息。我们看到这张图片单元地图中一共有 193 个图片单元，它们的尺寸都是  $64 \times 64$  像素，颜色深度是 24 位。

在将大位图导入到 Mappy 时，它会从位图的左上角开始抓取图片单元，从左到右、从上到下一格一格处理图像，直到整个图像都编入图片单元中。然后它使用图片单元编号来构造图片单元地图并将图片单元地图插入到编辑器中，于是这个地图就和原来的位图图像一样了。注意，创

建出来的图片单元地图至少和原来的位图一样大（本例中是  $1024 \times 768$ ），甚至更大。

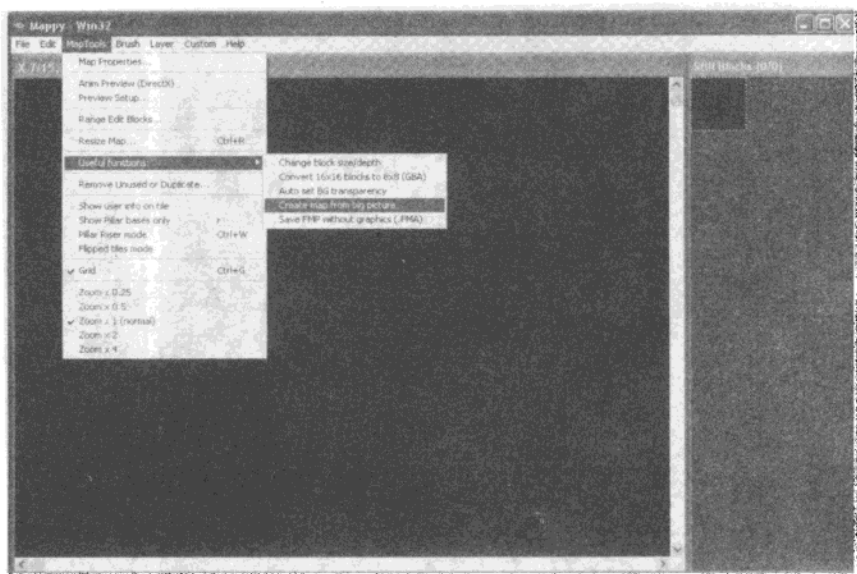


图 10-9 准备导入大的位图文件作为图片单元的来源

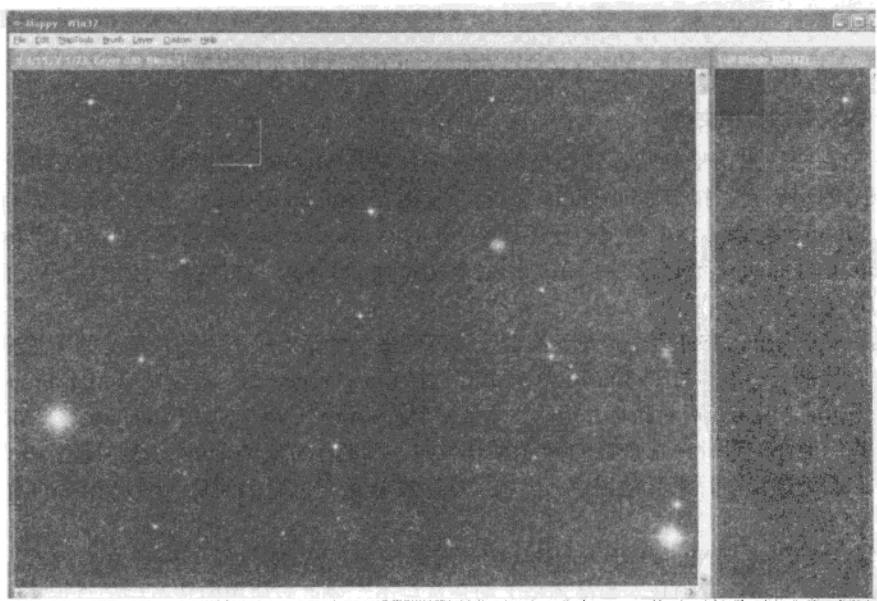


图 10-10 从大的太空照片中导入图片单元画板

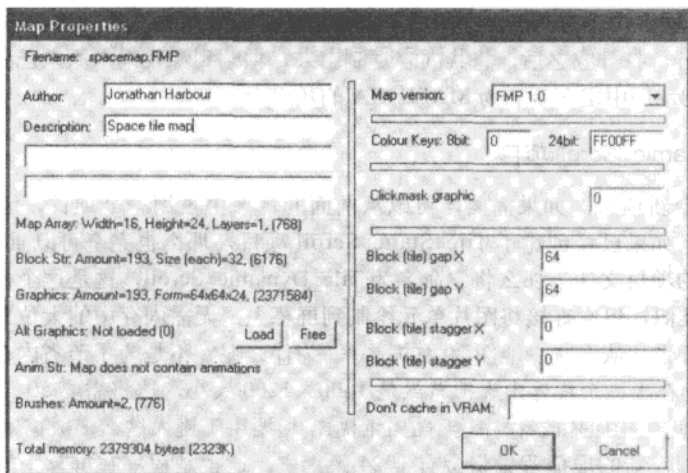


图 10-11 Map Properties 对话框显示图片单元地图的属性

## 2. 导出图片单元地图

我们先将图片单元地图以 Mappy 的本地文件格式保存，以便今后编辑。打开 File 菜单并选择 Save。将这个图片单元地图命名为 spacemap。Mappy 文件的默认扩展名是 .fmp。

现在，如果需要的话可以继续编辑图片单元地图。下面介绍将图片单元地图导出的方法。首先，打开 File 菜单并选择 Export 选项。系统显示 Export 对话框，如图 10-12 所示。按下列内容选择对话框上的选项：

- Map array as comma values only (? .CSV)
- Graphics Blocks as picture (? .BMP)
- 16 Blocks a row

使用这些选项 Mappy 将导出一个以出现在画板中的图片单元为顺序组成的新位图文件，也就意味着这个位图图像将可用于在游戏中绘制图片单元。注意，Mappy 会在画板中的第一个位置自动插入一块空白图片单元。这个单元必须保留在原位，因为图片单元地图值是从这块空白单元开始的（索引号为零）。前面已将导出文件命名为 spacemap。

单击 Ok 按钮，Mappy 将保存两个新文件以便我们使用：

- spacemap.csv
- spacemap.bmp

.csv 文件是以逗号分隔的值文件，它实际上以文本格式存储（可以使用记事本或者其他文本编辑器打开）。如果安装了 Microsoft Excel，则双击这个文件时它会尝试打开 .csv 文件，因

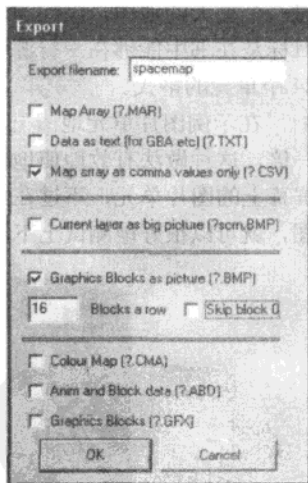


图 10-12 使用 Export 对话框将图片单元地图导出成文本文件

为 Excel 也使用这种格式作为基于文本的电子表格格式。如果需要，可以将它重命名为 spacemap.txt 以便更容易打开。打开它之后，将其内容复制出来，粘贴到源代码中原来已经存在的图片单元地图上（在本章的示例中定义于名为 MAPDATA 的数组中）。

### 10.2.3 Tile\_Dynamic\_Scroll 项目

下面创建一个新项目。如果需要，可以从前面的章节中重用个项目，因为这样库文件等都已经正确配置。如果已经创建了 Tile\_Static\_Scroll 项目，那么也完全可以重用这个项目。

如果创建新的项目文件，那么将它命名为 Tile\_Dynamic\_Scroll，这是这个程序的名称。这个程序与静态演示类似，但它直接将图片单元绘制到屏幕上，无需内存中的大位图。这个程序也将使用更小的虚拟背景，从而减少地图数组的尺寸。为什么呢？不是为了节省内存，而是为了让程序更可管理。因为在上一个程序中虚拟背景是 1600×1200 大小，这将需要 50 列宽、37 行深的图片单元来填满！对于地图编辑器程序这不是个问题，但要手工键入这么多数据则是太多了。

为了更可管理，新的虚拟背景将会有 1024 像素宽，这也是程序在屏幕上的宽度。这里是特意这么做的，因为动态滚动程序将模拟一个垂直卷动的射击街机游戏！这个程序的要点在于演示它的工作原理，而不是构建一个游戏引擎，所以目前不要担心精度问题。如果想键入这些值来创建一个更大的地图，没问题，干吧！这实际上会是个极佳的学习体验。为了方便读者（我的主要目标是在书中能够在一行源代码中打印完整的一行数字），本书将坚持使用宽 16 个图片单元、深 24 个单元的格式。

在示例图片单元地图上，通过复制整个图片单元地图值然后在末尾粘贴的方法将其尺寸增大一倍，这种做法有效地使地图尺寸加倍；否则就无法滚动它了。在这样的游戏中我们将来回滚动屏幕上的图片单元，不过在本示例中，滚动是通过鼠标来控制的。通过使用比屏幕高度更高的地图，就可以很好地测试上下卷动的功能了。图 10-13 显示了动态滚动演示程序的输出。



图 10-13 动态滚动程序演示对定义在 map 数组中的地图进行滚动

## Tile\_Dynamic\_Scroll 源代码

键入下面的动态卷动演示项目的源代码。这些代码位于 MyGame.cpp 文件中。

```

/*
    Beginning Game Programming, Third Edition
    MyGame.cpp
*/

#include "MyDirectX.h"
#include <sstream>
using namespace std;

const string APPTITLE = "Tile-Based Dynamic Scrolling";
const int SCREENW = 1024;
const int SCREENH = 768;

LPD3DXFONT font;

//settings for the scroller
const int TILEWIDTH = 64;
const int TILEHEIGHT = 64;
const int MAPWIDTH = 16;
const int MAPHEIGHT = 24;

//scrolling window size
const int WINDOWWIDTH = (SCREENW / TILEWIDTH) * TILEWIDTH;
const int WINDOWHEIGHT = (SCREENH / TILEHEIGHT) * TILEHEIGHT;

int ScrollX, ScrollY;
int SpeedX, SpeedY;
long start;
LPDIRECT3DSURFACE9 scrollbuffer=NULL;
LPDIRECT3DSURFACE9 tiles=NULL;

int MAPDATA[MAPWIDTH*MAPHEIGHT] = {
1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,
26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,
48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,
70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,
92,93,94,95,96,97,98,99,100,101,102,103,104,105,106,107,108,109,
110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,
126,127,128,129,130,131,132,133,134,135,136,137,138,139,140,141,
142,143,144,145,146,147,148,149,150,151,152,153,154,155,156,157,
158,159,160,161,162,163,164,165,166,167,168,169,170,171,172,173,
174,175,176,177,178,179,180,181,182,183,184,185,186,187,188,189,
190,191,192,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,
42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,
63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,
84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,102,103,
104,105,106,107,108,109,110,111,112,113,114,115,116,117,118,119,
120,121,122,123,124,125,126,127,128,129,130,131,132,133,134,135,
136,137,138,139,140,141,142,143,144,145,146,147,148,149,150,151,

```

```

152,153,154,155,156,157,158,159,160,161,162,163,164,165,166,167,
168,169,170,171,172,173,174,175,176,177,178,179,180,181,182,183,
184,185,186,187,188,189,190,191,192
};

```

```

bool Game_Init(HWND window)
{
    Direct3D_Init(window, SCREENW, SCREENH, false);
    DirectInput_Init(window);

    //create pointer to the back buffer
    d3ddev->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, &backbuffer);

    //create a font
    font = MakeFont("Arial", 24);

    //load the tile images
    tiles = LoadSurface("spacemap.bmp");
    if (!tiles) return false;

    //create the scroll buffer surface in memory, slightly bigger
    //than the screen
    const int SCROLLBUFFERWIDTH = SCREENW + TILEWIDTH * 2;
    const int SCROLLBUFFERHEIGHT = SCREENH + TILEHEIGHT * 2;

    HRESULT result = d3ddev->CreateOffscreenPlainSurface(
        SCROLLBUFFERWIDTH, SCROLLBUFFERHEIGHT,
        D3DFMT_X8R8G8B8, D3DPOL_DEFAULT,
        &scrollbuffer,
        NULL);
    if (result != S_OK) return false;

    start = GetTickCount();

    return true;
}

void Game_End()
{
    if (scrollbuffer) scrollbuffer->Release();
    if (tiles) tiles->Release();
    DirectInput_Shutdown();
    Direct3D_Shutdown();
}

//This function updates the scrolling position and speed
void UpdateScrollPosition()
{
    const int GAMEWORLDWIDTH = TILEWIDTH * MAPWIDTH;
    const int GAMEWORLDHEIGHT = TILEHEIGHT * MAPHEIGHT;

    //update horizontal scrolling position and speed
    ScrollX += SpeedX;

```



```

    if (ScrollX < 0)
    {
        ScrollX = 0;
        SpeedX = 0;
    }
    else if (ScrollX > GAMEWORLDWIDTH - WINDOWWIDTH)
    {
        ScrollX = GAMEWORLDWIDTH - WINDOWWIDTH;
        SpeedX = 0;
    }

    //update vertical scrolling position and speed
    ScrollY += SpeedY;
    if (ScrollY < 0)
    {
        ScrollY = 0;
        SpeedY = 0;
    }
    else if (ScrollY > GAMEWORLDHEIGHT - WINDOWHEIGHT)
    {
        ScrollY = GAMEWORLDHEIGHT - WINDOWHEIGHT;
        SpeedY = 0;
    }
}

//This function does the real work of drawing a single tile from the
//source image onto the tile scroll buffer
void DrawTile(
    LPDIRECT3DSURFACE9 source, // source surface image
    int tilenum,               // tile #
    int width,                 // tile width
    int height,                // tile height
    int columns,               // columns of tiles
    LPDIRECT3DSURFACE9 dest,   // destination surface
    int destx,                 // destination x
    int desty)                 // destination y
{
    //create a RECT to describe the source image
    RECT r1;
    r1.left = (tilenum % columns) * width;
    r1.top = (tilenum / columns) * height;
    r1.right = r1.left + width;
    r1.bottom = r1.top + height;

    //set destination rect
    RECT r2 = {destx, desty, destx + width, desty + height};

    //draw the tile
    d3ddev->StretchRect(source, &r1, dest, &r2, D3DTEXF_NONE);
}

//This function fills the tilebuffer with tiles representing
//the current scroll display based on scrollx/scrolly.

```



```

void DrawTiles()
{
    int tilenx, tiley;
    int columns, rows;
    int x, y;
    int tilenum;

    //calculate starting tile position
    tilenx = ScrollX / TILEWIDTH;
    tiley = ScrollY / TILEHEIGHT;

    //calculate the number of columns and rows
    columns = WINDOWWIDTH / TILEWIDTH;
    rows = WINDOWHEIGHT / TILEHEIGHT;

    //draw tiles onto the scroll buffer surface
    for (y=0; y<=rows; y++)
    {
        for (x=0; x<=columns; x++)
        {
            //retrieve the tile number from this position
            tilenum = MAPDATA[((tiley + y) * MAPWIDTH + (tilenx + x))];

            //draw the tile onto the scroll buffer
            DrawTile(tiles, tilenum, TILEWIDTH, TILEHEIGHT, 16, scrollbuffer,
                x*TILEWIDTH, y*TILEHEIGHT);
        }
    }
}

//This function draws the portion of the scroll buffer onto the back
//buffer according to the current "partial tile" scroll position.
void DrawScrollWindow(bool scaled = false)
{
    //calculate the partial sub-tile lines to draw using modulus
    int partialx = ScrollX % TILEWIDTH;
    int partially = ScrollY % TILEHEIGHT;

    //set dimensions of the source image as a rectangle
    RECT r1 = {partialx, partially, partialx+WINDOWWIDTH-1,
        partially+WINDOWHEIGHT-1};

    //set the destination rectangle
    RECT r2;
    if (scaled) {
        //use this line for scaled display
        RECT r = {0, 0, WINDOWWIDTH-1, WINDOWHEIGHT-1};
        r2 = r;
    }
    else {
        //use this line for non-scaled display

```

PDF  
PDG

```

    RECT r = {0, 0, SCREENW-1, SCREENH-1};
    r2 = r;
}

//draw the "partial tile" scroll window onto the back buffer
d3ddev->StretchRect(scrollbuffer, &r1, backbuffer, &r2,
    D3DTEXF_NONE);
}

void Game_Run(HWND window)
{
    if (!d3ddev) return;
    DirectInput_Update();
    d3ddev->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
        D3DCOLOR_XRGB(0,0,100), 1.0f, 0);

    //scroll based on key or controller input
    if (Key_Down(DIK_DOWN) || controllers[0].sThumbLY < -2000)
        ScrollY += 1;

    if (Key_Down(DIK_UP) || controllers[0].sThumbLY > 2000)
        ScrollY -= 1;

    //keep the game running at a steady frame rate
    if (GetTickCount() - start >= 30)
    {
        //reset timing
        start = GetTickCount();

        //update the scrolling view
        UpdateScrollPosition();

        //start rendering
        if (d3ddev->BeginScene())
        {
            //draw tiles onto the scroll buffer
            DrawTiles();

            //draw the scroll window onto the back buffer
            DrawScrollWindow();
            spriteobj->Begin(D3DXSPRITE_ALPHABLEND);

            std::ostringstream oss;
            oss << "Scroll Position = " << ScrollX << ", " << ScrollY;
            FontPrint(font, 0, 0, oss.str());

            spriteobj->End();

            //stop rendering
            d3ddev->EndScene();
            d3ddev->Present(NULL, NULL, NULL, NULL);
        }
    }
}

```



```
//to exit
if (KEY_DOWN(VK_ESCAPE) ||
    controllers[0].wButtons & XINPUT_GAMEPAD_BACK)
    gameover = true;
}
```

要一下消化这个程序有点困难，而且这里也没有非常认真地讲解每个细节，因为现在必须转换到 3D 了，不能在 2D 图形上花更多时间！不过这段代码是可重用的，很容易就可用它来构建一个卷动的街机游戏。只需让卷动器自己移动（无要求用户输入），然后在上面添加一些精灵，很快地，我们就有了一个卷动的街机游戏！

### 10.3 基于位图的卷动

使用 Direct3D 表面对象完成背景卷动还有一种方法，使用全位图卷动。它需要进行一些非常复杂的编程来生成一种算法，实现将单个位图卷绕到卷动窗口中而无需使用图片单元（我们在前两个示例程序中使用过）。这样代码不仅会变得复杂，而且使用图片卷绕来渲染基于位图的卷动器也比较慢。

#### 10.3.1 基于位图的卷动理论

为了让基于位图的卷动尽可能高效地执行，建议让源图片保持和屏幕一样的尺寸。这样也许会使卷动器不能和支持任何分辨率的卷动器一样通用，但如果真的需要这种通用性，走这条路也不困难（需要牺牲一些性能，因为需要将结果的卷动缓冲区缩放放到屏幕上）。

那么，对于初学者，要做的就是确认源位图与屏幕尺寸相同。然后，必须要做的是创建一个更大的卷动缓冲区，其大小为源图像（例如屏幕尺寸）的 4 倍（4×）。这可以通过在内存中创建一个大的 Direct3D 表面来实现，然后将源图像粘贴到卷动缓冲区的四个角。这会占用大量内存，但却可以以任何方向卷动！图 10-14 展示了在源图像粘贴 4 次之后的卷动缓冲区的样子（分割 4 个图像的白线只是展示用的，实际的卷动缓冲区没有这样的分割线）。

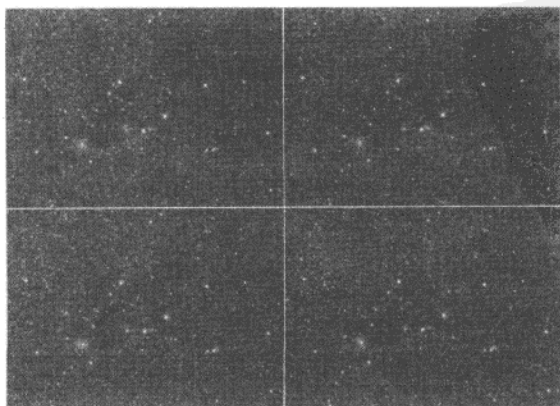


图 10-14 卷动缓冲区是一个 4 倍于源位图尺寸的表面

执行这种类型的卷动包括这些主要步骤：

- 1) 装载卷动器所用的源位图。
- 2) 创建源位图 4 倍尺寸的卷动缓冲区。
- 3) 将源位图复制到卷动缓冲区的 4 个角。
- 4) 渲染卷动缓冲区中与卷动位置有关的部分。

### 10.3.2 位图卷动演示

Bitmap\_Scrolling 程序演示了基于位图的卷动，如图 10-15 所示。这个程序有趣之处在于，和前面两个项目不同，我们可以以任何方向卷动它，不仅仅只是水平或者垂直方向，而是任何斜线角度都可以。

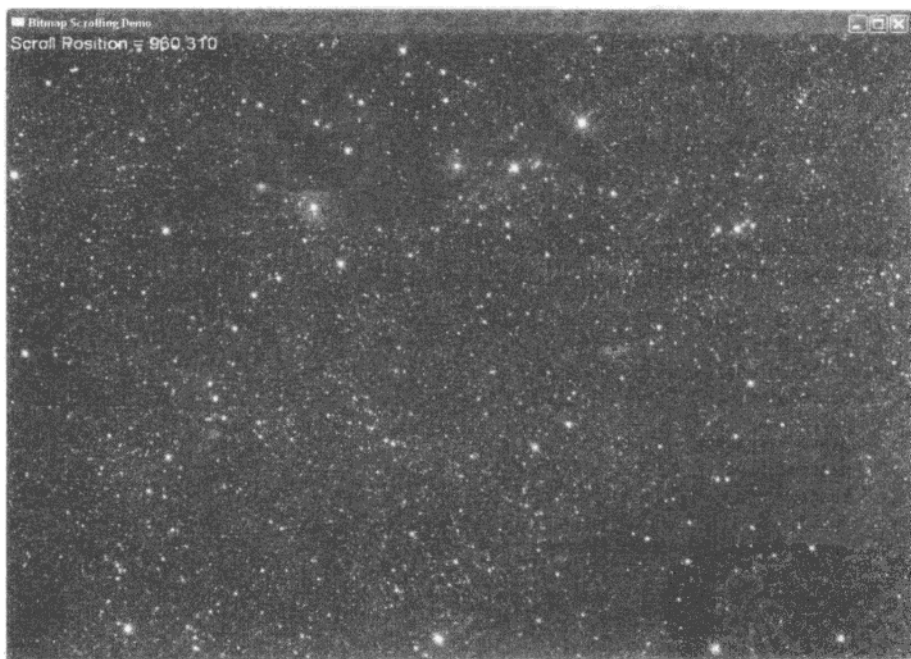


图 10-15 Bitmap\_Scrolling 演示（感谢 Space Telescope Science Institute 提供太空图片，[www.scsti.edu](http://www.scsti.edu)）

```
/*
    Beginning Game Programming, Third Edition
    MyGame.cpp
*/

#include "MyDirectX.h"
#include <sstream>
using namespace std;
```

```
const string APPTITLE = "Bitmap Scrolling Demo";
const int SCREENW = 1024;
const int SCREENH = 768;

const int BUFFERW = SCREENW * 2;
const int BUFFERH = SCREENH * 2;

LPDIRECT3DSURFACE9 background = NULL;

LPD3DXFONT font;

double scrollx=0, scrolly=0;

bool Game_Init(HWND window)
{
    Direct3D_Init(window, SCREENW, SCREENH, false);
    DirectInput_Init(window);

    //create a font
    font = MakeFont("Arial", 24);

    //load background
    LPDIRECT3DSURFACE9 image = NULL;
    image = LoadSurface("space2.bmp");
    if (!image) return false;

    //create background
    HRESULT result =
    d3ddev->CreateOffscreenPlainSurface(
        BUFFERW,
        BUFFERH,
        D3DFMT_X8R8G8B8,
        D3DPPOOL_DEFAULT,
        &background,
        NULL);
    if (result != D3D_OK) return false;

    //copy image to upper left corner of background
    RECT source_rect = {0, 0, 1024, 768 };
    RECT dest_ul = { 0, 0, 1024, 768 };
    d3ddev->StretchRect(image, &source_rect, background, &dest_ul, D3DTEXF_NONE);

    //copy image to upper right corner of background
    RECT dest_ur = { 1024, 0, 1024*2, 768 };
    d3ddev->StretchRect(image, &source_rect, background, &dest_ur, D3DTEXF_NONE);

    //copy image to lower left corner of background
    RECT dest_ll = { 0, 768, 1024, 768*2 };
    d3ddev->StretchRect(image, &source_rect, background, &dest_ll, D3DTEXF_NONE);

    //copy image to lower right corner of background
    RECT dest_lr = { 1024, 768, 1024*2, 768*2 };
    d3ddev->StretchRect(image, &source_rect, background, &dest_lr, D3DTEXF_NONE);
}
```

```

//get pointer to the back buffer
d3ddev->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO, &backbuffer);

//remove scratch image
image->Release();

return true;
}

void Game_Run(HWND window)
{
    if (!d3ddev) return;
    DirectInput_Update();
    d3ddev->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
        D3DCOLOR_XRGB(0,0,100), 1.0f, 0);

    if (Key_Down(DIK_UP) || controllers[0].sThumbLY > 2000)
        scrolly -= 1;

    if (Key_Down(DIK_DOWN) || controllers[0].sThumbLY < -2000)
        scrolly += 1;

    if (Key_Down(DIK_LEFT) || controllers[0].sThumbLX < -2000)
        scrollx -= 1;

    if (Key_Down(DIK_RIGHT) || controllers[0].sThumbLX > 2000)
        scrollx += 1;

    //keep scrolling within boundary
    if (scrolly < 0)
        scrolly = BUFFERH - SCREENH;
    if (scrolly > BUFFERH - SCREENH)
        scrolly = 0;
    if (scrollx < 0)
        scrollx = BUFFERW - SCREENW;
    if (scrollx > BUFFERW - SCREENW)
        scrollx = 0;

    if (d3ddev->BeginScene())
    {
        RECT source_rect = {scrollx, scrolly, scrollx+1024, scrolly+768};
        RECT dest_rect = {0, 0, 1024, 768};

        d3ddev->StretchRect(background, &source_rect, backbuffer,
            &dest_rect, D3DTEXF_NONE);

        spriteobj->Begin(D3DXSPRITE_ALPHABLEND);

        std::ostringstream oss;
        oss << "Scroll Position = " << scrollx << ", " << scrolly;
        FontPrint(font, 0, 0, oss.str());

        spriteobj->End();
        d3ddev->EndScene();
    }
}

```

```

        d3ddev->Present(NULL, NULL, NULL, NULL);
    }

    if (KEY_DOWN(VK_ESCAPE)) gameover = true;
    if (controllers[0].wButtons & XINPUT_GAMEPAD_BACK)
        gameover = true;
}

void Game_End()
{
    background->Release();
    font->Release();
    DirectInput_Shutdown();
    Direct3D_Shutdown();
}

```

## 10.4 你所学到的

本章我们学习的是背景卷动。学习了如何在游戏中创建它、使用它。使用图片单元来创建一个卷动的游戏世界绝不是一个简单的课题！有以下几个要点：

- 如何创建虚拟卷动缓冲区。
- 如何使用 Mappy 来创建图片单元地图。
- 如何动态在屏幕上绘制图片单元。

## 10.5 复习测验

以下复习测验题将考验读者对本章所讲内容的理解。

- 1) 在静态卷动程序中所用的虚拟卷动缓冲区分辨率是多少？
- 2) 同样的，在动态卷动程序中所用的虚拟卷动缓冲区分辨率是多少？
- 3) 在两个示例程序中，图片单元绘制代码之间有什么不同？
- 4) 如何使用 Mappy 为巨大的有数千个图片单元的游戏关创建一个图片单元地图？
- 5) 在动态绘制图片单元的游戏关中，地图尺寸的有效限制是多少？
- 6) Mappy 本地游戏关文件的文件扩展名是什么？
- 7) 为了将 Mappy 游戏关文件转换为可在 DirectX 程序中使用的形式，要进行哪种类型的导出？
- 8) 对于位图卷动器，将源背景图片位块传输到卷动缓冲区上要进行多少次？
- 9) Mappy 用于表示游戏关里的各个图片单元的术语是什么？
- 10) 如果想创建一个与老 Mario 平台游戏相似的游戏，将使用位图卷动器还是图片单元卷动器？

## 10.6 自己动手

以下习题将考验读者对本章知识的记忆能力。

习题1 动态卷动程序肯定还有大量潜力，我们在这里只不过是做了肤浅的研究而已！看看你自己能否让程序自动卷动图片单元地图而无需用户输入。

习题2 动态卷动程序看起来几乎就像一个初级的不带自动卷动的游戏一样，那么，何不更进一步。装载一个表示宇宙飞船的精灵，然后将其绘制在卷动器之上的屏幕上。而后，允许游戏者使用箭头键左右移动精灵。





## 第 11 章 播放音频

音频对于游戏是至关重要的！为了让游戏者沉浸于游戏的虚幻体验中（称为“怀疑暂停”，suspension of disbelief），音效和音乐起了极大的作用，而且在游戏者中建立了情绪反应。如果将音频从游戏中移除，那么游戏者将会有不同的响应。而如果同一个游戏有动态的、强劲的音响效果和恰当的背景音乐，那么整个体验都会改变。本章将介绍使用 DirectSound 在听觉上增强游戏的方法。有才华的游戏设计者会使用音频来影响游戏者的心情。本章在探究 DirectSound 的同时，也将利用这一机会进一步处理精灵碰撞。

本章将学到：

- 如何初始化 DirectSound。
- 如何从波形文件中装载音频。
- 如何使用混音（mixing）效果来播放音效。
- 如何使用混音效果来循环播放音效。

### 11.1 使用 DirectSound

DirectSound 是 DirectX 中为游戏处理所有声音输出的组件，它有一个多通道的声音混合器。基本上，只需告诉 DirectSound 要播放的声音，它会处理所有的细节（包括将这个声音与当前播放中的声音混合在一起）。使用 DirectSound 创建、初始化、装载及播放波形文件所需的代码比起我们在前几章所学的位图和精灵代码来说要复杂一些。所以，为了避免重复劳动，这里将把使用 Microsoft 自己的 DirectSound 包装器的方法介绍给大家。

作为程序员，使用包装器违背了我的本性，因为我总想知道我所用的代码中的一切，而且经常选择使用我自己编写的代码而不是别人的代码。不过，有时候，如果考虑时间因素，我们必须妥协并且使用已有的东西。毕竟，DirectX 本身就是一些人编写的游戏库，在游戏编程中坚持严格的哲学却让自己的进度慢吞吞，显然不合情理。如果你写的大部分都是 C 代码，如本书中的代码，这不会有问题，因为偶尔我们会需要稍微深入到 C++ 中，以便重用代码。就目前而言，我们将使用 DirectSound Utility 类，但这里不准备讲解其工作原理的细节。你可以把它当成 SDK，就如 DirectX 本身一样——有许多代码我们不理解，但只要这些代码能工作，就可以编写游戏而无需担忧它们。

DirectX SDK 包括一个称为 DXUTsound 的实用工具库。我们不准备使用它，因为它需要的支持文件太多。我们将使用一个老一点的版本，它来自以前的 DirectX 9.0c 版，我对它一直难以割舍。老的 DirectSound 的 DXUT 版可以在名为 dsutil.cpp 和 dsutil.h 的这一对文件中找到。

Microsoft 在其 DirectX Utility 库（DXUT）上的作为实在是太不可预测了。对于 DirectSound 助手函数和类（CSoundManager、CSound 和 CWaveFile）而言，一致性的问题尤为尖锐，而这类是我们使用 DirectSound 来装载和播放波形文件需要的。

在版本老一些的 DirectX 中，这些助手类位于 dsutil.h 和 dsutil.cpp 中。在后来的 DirectX 版本中，它们与 DXUTsound.h 和 DXUTsound.cpp 合并在一起。最新版的 DirectX（本书编写时）将这些类藏到了另外一组文件中：SDKsound.h、SDKsound.cpp 和 SDKwavefile.h。不一致性仍然在继续！在 SDKsound.h 文件中的头注释里它被称为 DXUTsound.h！

由于这是一个重复发生的问题，所以我创建了一对新的音频文件以便我们自己使用，它们的名称为 DirectSound.h 和 DirectSound.cpp。这些文件包含来自老的 dsutil 文件的源代码（因为这些代码即使在最新的 DirectX SDK 中也没有更改）。我们会把这些文件添加到可重用的 DirectX Project 模板中。

我们感兴趣的是定义在 SDKsound（以前的 DXUTsound）中的三个类：

CStateManager	主 DirectSound 设备
CSound	用于创建 DirectSound 缓冲区
CWaveFile	帮助将波形文件装载到 CSound 缓冲区中

### 11.1.1 初始化 DirectSound

为使用 DirectSound，首先要做的是创建 CStateManager 类的实例（也就是创建“类”的“对象”）。

```
CStateManager *dsound = new CStateManager();
```

下一步调用 Initialize 函数来初始化 DirectSound 管理器：

```
dsound->Initialize(window_handle, DSSCL_PRIORITY);
```

第一个参数是程序的窗口句柄，而第二个参数指定 DirectSound 的协作级别，一共有三个选择：DSSCL\_NORMAL。与其他程序共享声音设备。

DSSCL\_PRIORITY。获取对声音设备的更高优先级（建议游戏使用）。

DSSCL\_WRITEPRIMARY。提供对主声音缓冲区的修改访问权限。

最常用的协作级别是 DSSCL\_PRIORITY，它为游戏程序提供比其他可能正在运行的程序更高的声音设备优先级。

在初始化 DirectSound 之后，我们必须设置音频缓冲区格式。通常这不是我们需要参与的事情，但如果我们愿意的话的确有个选项可以更改声音混音器的内部格式（用于调整音频回放质量）。在下面这行代码中，将音频缓冲区配置为立体声、22kHz、16 位。如果制作一款要求达到 CD 音频质量的游戏，那么就需要将这个设置抬高一两个级别（例如，CD 质量的音频大致是 44 kHz，但大多数波形文件以更低的码率编码）。

```
dsound->SetPrimaryBufferFormat(2, 22050, 16);
```

### 11.1.2 创建声音缓冲区

在初始化了 DirectSound 管理器（通过 CStateManager）之后，通常将游戏所需的所有音响

效果装载进来。我们通过使用如下定义的 CSound 指针变量来访问音响效果：

```
CSound *wave;
```

我们创建的 CSound 对象是名为 LPDIRECTSOUNDBUFFER8 的第二声音缓冲区的包装器，由于存在实用工具类，因此我们无需自己来编程。

### 11.1.3 装载波形文件

可以把由 DirectSound 创建和管理的声音混音器当成声音的主缓冲区。如 Direct3D 一样，主缓冲区是输出发生的地方。只不过在 DirectSound 的情况中，第二缓冲区是声音数据而不是位图数据，通过调用 Play 函数来播放声音（很快就会讲解）。

将波形文件装载到 DirectSound 的第二缓冲区中只涉及对一个函数的调用，这里无需列出许多页的代码清单来初始化第二缓冲区，打开波形文件，读入内存然后配置所有的参数。我们创建的 CSoundManager 对象有装载波形文件所需的函数，它名为 Create：

```
HRESULT Create(
    CSound** ppSound,
    LPTSTR strWaveFileName,
    DWORD dwCreationFlags = 0,
    GUID guid3DAlgorithm = GUID_NULL,
    DWORD dwNumBuffers = 1
);
```

第一个参数指定用于新装载的波形声音的 CSound 对象。第二个参数是文件名。剩下的参数可使用默认值，也就是说实际只需要使用两个参数来调用这个函数。以下是示例：

```
dsound->Create(&wave, "snicker.wav");
```

### 11.1.4 播放声音

我们可以自由地播放声音，想播就播，无需担心声音混音、声音回放结束或者任何其他细节，因为 DirectSound 本身会为我们处理所有这些细节。在 CSound 类本身中有个名为 Play 的函数为我们播放声音。以下是这个函数的代码：

```
HRESULT Play(
    DWORD dwPriority = 0,
    DWORD dwFlags = 0,
    LONG lVolume = 0,
    LONG lFrequency = -1,
    LONG lPan = 0
);
```

第一个参数是优先级，这是个高级选项，必须总是设置为零。第二个参数指定是否想让声音循环播放，也就是每次到达波形数据的末尾时它会从头重新开始，继续播放。如果想循环播放声音，使用 DSBPLAY\_LOOPING 值。最后三个参数指定声音的音量、频率和左右声道平衡，这些值也可使用默认值，不过如果需要的话也可自己设置。

以下这个示例给出了调用这一函数进行正常音频回放的通常方法。可自己改写这些参数，如果想使用默认值也可完全保留原样不变。

```
wave->Play();
```

以下是使用循环的方法：

```
wave->Play(0, DSBPLAY_LOOPING);
```

要停止正在播放的声音，可使用 Stop 函数。这个函数对于循环声音尤其有用，因为循环播放会永远继续下去，除非通过不使用循环参数再次播放这个声音的方法来停止或重置声音。

```
HRESULT Stop();
```

这个函数的使用示例实在是简单：

```
wave->Stop();
```

**建议** 虽然对我们而言 DirectSound 已经足够用了，但游戏项目还可以使用更好的音频引擎。笔者推荐 Firelight Technologies 的 FMOD。对于非商业使用的情况这个软件是免费的：[www.fmod.org](http://www.fmod.org)。笔者个人在多数项目和更高级的书籍中都使用这一音频库。

## 11.2 测试 DirectSound

下面编写一个简单的示例来尝试本章所学的 DirectSound 代码的编写方法。由于 DirectSound 是个新组件，所以在 MyDirectX.h 和 MyDirectX.cpp 文件中还没有对它的支持，这是我们需要调整的。在配置了新项目并且添加了 DirectSound 代码之后，将讲解能让一组撞球模样的球在四个缓冲器之间来回互相撞击并且在每次撞击时播放声音的程序，且以其代码作为音频演示。Play\_Sound 程序如图 11-1 所示。

Play\_Sound 程序在球的移动及在缓冲器上的反弹（或者互相撞击）上没有使用真实的物理轨迹，它只是按照每个对象的位置大致反弹。所以球的反弹效果不是非常吸引人，但毕竟这不是一个物理演示。

令人惊讶的是，在 Play\_Sound 程序所有的代码行中，涉及音频回放的只有非常少的几行！不过这不是件坏事，这反而是我们追求的目标：让音频接口尽可能简单、无痛苦，正是我们要实现的。在运行过程中，这个 Play\_Sound 程序以颇为有趣的方式结束！你是不是想从这里开始编写一个撞球游戏了？

### 11.2.1 创建项目

如果只想打开并试试 Play\_Sound 项目，在 CD-ROM 中有完成的版本。当然，也可以打开同

样是在本章文件夹中的 DirectX\_Project 模板，配置它然后添加 Play\_Sound 程序的源代码。

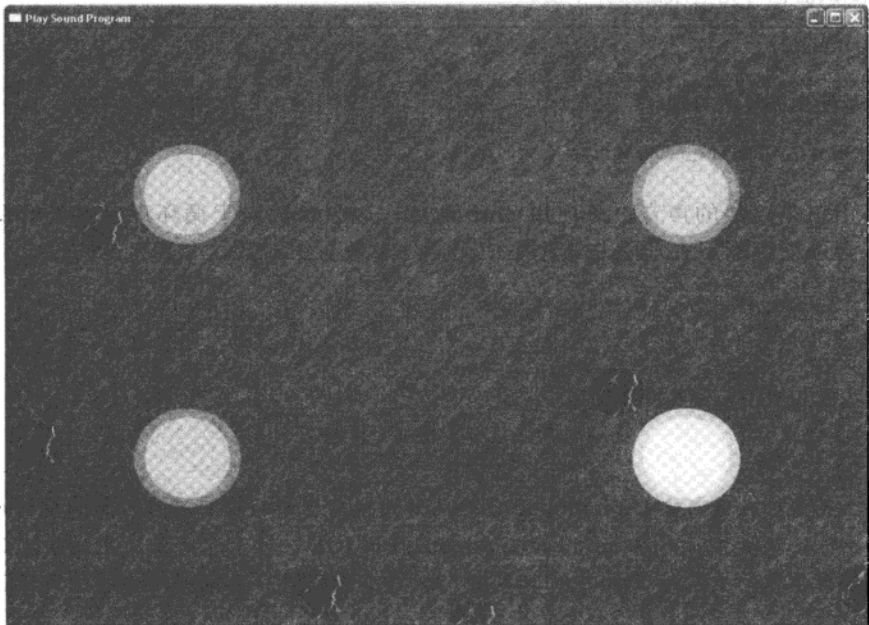


图 11-1 Play\_Sound 程序演示了 DirectSound 的使用方法

如果想从头开始创建新的项目，可按照如下基本方法进行：

- 1) 运行 Visual C++。
- 2) 打开 File 菜单并选择 New，打开 New 对话框。
- 3) 确认选择了 Projects 选项卡。选择 Win32 Application 项目类型，然后在项目名称中输入 Play\_Sound。
- 4) 单击 OK 按钮关闭对话框并创建新项目。和往常一样，不要让 Visual C++ 为我们添加任何文件。

从现在开始我们需要在 DirectX\_Project 模板中包括两个新文件，它们马上要在 Play\_Sound 程序中使用。这两个文件包含 Microsoft 使用 DirectSound 的代码，它们由许多支持类组成：CSoundManager、CSound 和 CWaveFile。这些文件包含在 CD-ROM 中，由于代码太长就不在这里印刷出来了。这两个文件是：

- DirectSound.h
- DirectSound.cpp

这些文件不在 DirectX SDK 中，它们来自于前面讨论的 DXUT 文件，可以无需 DXUT 独立编译。确认在项目中添加了这些文件。接下来，需要在 MyDirectX.h 和 MyDirectX.cpp 中添加新的能够方便地使用音频类的音频函数。

在把这些文件复制到新的项目文件夹之后，可以通过在 Visual C++ 中打开 Project 菜单并选择 Add Existing Item 菜单项将它们添加到项目中。在系统随后弹出的 Add Existing Item 对话框中选择将 DirectSound.h 和 DirectSound.cpp 添加到项目中。在添加了这些文件之后，项目应包含这些文件：

- DirectSound.cpp
- DirectSound.h
- MyDirectX.cpp
- MyDirectX.h
- MyGame.cpp
- MyWindows.cpp

要想检验项目是否正确配置，参考图 11-2，它显示了装载了所有所需文件的 Solution Explorer。

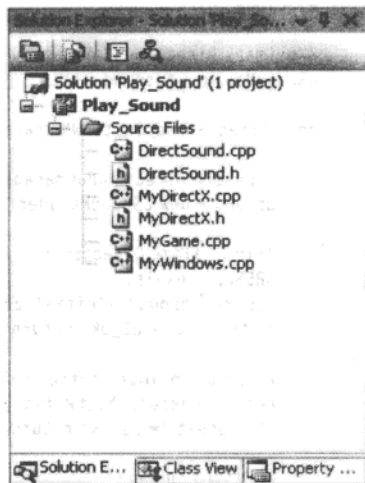


图 11-2 框架文件已经添加到了项目中

### 11.2.2 修改 MyDirectX 文件

好了，这是个很长的过程，但如果一路按部就班执行下来，那么现在应该有了一个可以编译的项目了。遗憾的是，game.h 和 game.cpp 文件包含的是上一个项目的代码，它们和 DirectSound 没有一点关系！不过，方便的是，这些文件已经位于项目中了，我们只需打开它们然后替换代码即可。

#### 1. MyDirectX.h 的添加内容

只需在 MyDirectX.h 中添加如下新代码即可在我们已经构建的（并且保存在每章的 DirectX\_ Project 模板中）框架中提供对 DirectSound 的支持。

```
//DirectSound code added in chapter 11
#include "DirectSound.h"

#pragma comment(lib,"dsound.lib")
#pragma comment(lib,"dxerr9.lib")

//primary DirectSound object
extern CSoundManager *dsound;

//function prototypes
bool DirectSound_Init(HWND hwnd);
void DirectSound_Shutdown();
CSound *LoadSound(string filename);
void PlaySound(CSound *sound);
void LoopSound(CSound *sound);
void StopSound(CSound *sound);
```

#### 2. MyDirectX.cpp 的添加内容

将下列代码添加到 MyDirectX.cpp 中：

```
// New DirectSound code

#include "DirectSound.h"

//primary DirectSound object
CSoundManager *dsound = NULL;

bool DirectSound_Init(HWND hwnd)
{
    //create DirectSound manager object
    dsound = new CSoundManager();

    //initialize DirectSound
    HRESULT result;
    result = dsound->Initialize(hwnd, DSSCL_PRIORITY);
    if (result != DS_OK) return false;

    //set the primary buffer format
    result = dsound->SetPrimaryBufferFormat(2, 22050, 16);
    if (result != DS_OK) return false;

    //return success
    return true;
}

void DirectSound_Shutdown()
{
    if (dsound) delete dsound;
}

CSound *LoadSound(string filename)
{
    HRESULT result;
    //create local reference to wave data
    CSound *wave = NULL;

    //attempt to load the wave file
    char s[255];
    sprintf(s, "%s", filename.c_str());
    result = dsound->Create(&wave, s);
    if (result != DS_OK) wave = NULL;

    //return the wave
    return wave;
}

void PlaySound(CSound *sound)
{
    sound->Play();
}

void LoopSound(CSound *sound)
{
    sound->Play(0, DSBPLAY_LOOPING);
}
```



```

}

void StopSound(CSound *sound)
{
    sound->Stop();
}

```

### 11.2.3 修改 MyGame.cpp

既然 DirectSound 助手文件已经添加到了项目中，DirectSound 助手函数已经添加到了 MyDirect X 文件中，那么我们可以很容易地装载并播放音频文件了。以下是 Play\_Sound 示例程序的完整源代码。其中关键的音频代码加黑显示以便读者参考。

```

#include "MyDirectX.h"
using namespace std;

const string APPTITLE = "Play Sound Program";
const int SCREENW = 1024;
const int SCREENH = 768;
LPDIRECT3DTEXTURE9 ball_image = NULL;
LPDIRECT3DTEXTURE9 bumper_image = NULL;
LPDIRECT3DTEXTURE9 background = NULL;

//balls
const int NUMBALLS = 10;
SPRITE balls[NUMBALLS];

//bumpers
SPRITE bumpers[4];

//timing variable
DWORD screentimer = timeGetTime();
DWORD coretimer = timeGetTime();
DWORD bumpertimer = timeGetTime();

//the wave sounds
CSound *sound_bounce = NULL;
CSound *sound_electric = NULL;

bool Game_Init(HWND window)
{
    srand(time(NULL));

    //initialize Direct3D
    if (!Direct3D_Init(window, SCREENW, SCREENH, false))
    {
        MessageBox(window, "Error initializing Direct3D", APPTITLE.c_str(), 0);
        return false;
    }

    //initialize DirectInput

```



```
if (!DirectInput_Init(window))
{
    MessageBox(window, "Error initializing DirectInput",
        APPTITLE.c_str(), 0);
    return false;
}

//initialize DirectSound
if (!DirectSound_Init(window))
{
    MessageBox(window, "Error initializing DirectSound",
        APPTITLE.c_str(), 0);
    return false;
}

//load the background image
background = LoadTexture("craters.tga");
if (!background)
{
    MessageBox(window, "Error loading craters.tga", APPTITLE.c_str(), 0);
    return false;
}

//load the ball image
ball_image = LoadTexture("lightningball.tga");
if (!ball_image)
{
    MessageBox(window, "Error loading lightningball.tga",
        APPTITLE.c_str(), 0);
    return false;
}

//load the bumper image
bumper_image = LoadTexture("bumper.tga");
if (!ball_image)
{
    MessageBox(window, "Error loading bumper.tga", APPTITLE.c_str(), 0);
    return false;
}

//set the balls' properties
for (int n=0; n<NUMBALLS; n++)
{
    balls[n].x = (float)(rand() % (SCREENW-200));
    balls[n].y = (float)(rand() % (SCREENH-200));
    balls[n].width = 64;
    balls[n].height = 64;
    balls[n].velx = (float)(rand() % 6 - 3);
    balls[n].vely = (float)(rand() % 6 - 3);
}

//set the bumpers' properties
for (int n=0; n<4; n++)
{

```

```

        bumpers[n].width = 128;
        bumpers[n].height = 128;
        bumpers[n].columns = 2;
        bumpers[n].frame = 0;
    }
    bumpers[0].x = 150;
    bumpers[0].y = 150;
    bumpers[1].x = SCREENW-150-128;
    bumpers[1].y = 150;
    bumpers[2].x = 150;
    bumpers[2].y = SCREENH-150-128;
    bumpers[3].x = SCREENW-150-128;
    bumpers[3].y = SCREENH-150-128;

    //load bounce wave file
    sound_bounce = LoadSound("step.wav");
    if (!sound_bounce)
    {
        MessageBox(window, "Error loading step.wav", APPTITLE.c_str(), 0);
        return false;
    }

    return true;
}

void rebound(Sprite &spritel, Sprite &sprite2)
{
    float centerx1 = spritel.x + spritel.width/2;
    float centery1 = spritel.y + spritel.height/2;

    float centerx2 = sprite2.x + sprite2.width/2;
    float centery2 = sprite2.y + sprite2.height/2;

    if (centerx1 < centerx2)
    {
        spritel.velx = fabs(spritel.velx) * -1;
    }
    else if (centerx1 > centerx2)
    {
        spritel.velx = fabs(spritel.velx);
    }

    if (centery1 < centery2)
    {
        spritel.vely = fabs(spritel.vely) * -1;
    }
    else {
        spritel.vely = fabs(spritel.vely);
    }

    spritel.x += spritel.velx;
    spritel.y += spritel.vely;
}

```

```

void Game_Run(HWND window)
{
    int n;

    if (!d3ddev) return;
    DirectInput_Update();
    d3ddev->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
        D3DCOLOR_XRGB(0,0,100), 1.0f, 0);

    /*
     * slow ball movement
     */
    if (timeGetTime() > coretimer + 10)
    {
        //reset timing
        coretimer = GetTickCount();

        int width = balls[0].width;
        int height = balls[0].height;

        //move the ball sprites
        for (n=0; n<NUMBALLS; n++)
        {
            balls[n].x += balls[n].velx;
            balls[n].y += balls[n].vely;

            //warp the ball at screen edges
            if (balls[n].x > SCREENW)
            {
                balls[n].x = -width;
            }
            else if (balls[n].x < -width)
            {
                balls[n].x = SCREENW+width;
            }

            if (balls[n].y > SCREENH+height)
            {
                balls[n].y = -height;
            }
            else if (balls[n].y < -height)
            {
                balls[n].y = SCREENH+height;
            }
        }
    }

    //reset bumper frames
    if (timeGetTime() > bumpertimer + 250)
    {
        bumpertimer = timeGetTime();
        for (int bumper=0; bumper<4; bumper++)
        {

```



```

        bumpers[bumper].frame = 0;
    }
}

/*
 * check for ball collisions with bumpers
 */
for (int ball=0; ball<NUMBALLS; ball++)
{
    for (int bumper=0; bumper<4; bumper++)
    {
        if (CollisionD(balls[ball], bumpers[bumper]))
        {
            rebound(balls[ball], bumpers[bumper]);
            bumpers[bumper].frame = 1;
            PlaySound(sound_bounce);
        }
    }
}

/*
 * check for sprite collisions with each other
 * (as fast as possible-with no time limiter)
 */
for (int one=0; one<NUMBALLS; one++)
{
    for (int two=0; two<NUMBALLS; two++)
    {
        if (one != two)
        {
            if (CollisionD(balls[one], balls[two]))
            {
                while (CollisionD(balls[one], balls[two]))
                {
                    //rebound ball one
                    rebound(balls[one], balls[two]);

                    //rebound ball two
                    rebound(balls[two], balls[one]);
                }
            }
        }
    }
}

/*
 * slow rendering to approximately 60 fps
 */
if (timeGetTime() > screentimer + 14)
{
    screentimer = GetTickCount();
}

```



```
//start rendering
if (d3ddev->BeginScene())
{
    //start sprite handler
    spriteobj->Begin(D3DXSPRITE_ALPHABLEND);

    //draw background
    Sprite_Transform_Draw(background, 0, 0, SCREENW, SCREENH);

    //draw the balls
    for (n=0; n<NUMBALLS; n++)
    {
        Sprite_Transform_Draw(ball_image,
            balls[n].x, balls[n].y,
            balls[n].width, balls[n].height);
    }

    //draw the bumpers
    for (n=0; n<4; n++)
    {
        Sprite_Transform_Draw(bumper_image,
            bumpers[n].x,
            bumpers[n].y,
            bumpers[n].width,
            bumpers[n].height,
            bumpers[n].frame,
            bumpers[n].columns);
    }

    //stop drawing
    spriteobj->End();

    //stop rendering
    d3ddev->EndScene();
    d3ddev->Present(NULL, NULL, NULL, NULL);
}

//exit with escape key or controller Back button
if (KEY_DOWN(VK_ESCAPE)) gameover = true;
if (controllers[0].wButtons & XINPUT_GAMEPAD_BACK) gameover = true;
}

void Game_End()
{
    if (ball_image) ball_image->Release();
    if (bumper_image) bumper_image->Release();
    if (background) background->Release();
    if (sound_bounce) delete sound_bounce;

    DirectSound_Shutdown();
    DirectInput_Shutdown();
    Direct3D_Shutdown();
}
```

### 11.3 你所学到的

本章讲解了使用包含在 DirectX SDK 中的一些相对简单的 DirectSound 支持例程来简化 DirectSound 编程的方法。有以下几个要点：

- 如何初始化 DirectSound 对象。
- 如何将波形文件装载到声音缓冲区中。
- 如何带与不带循环地播放和停止声音。
- 一些关于声音混音的知识。
- 在一个有许多文件的项目上获得了一些实践。
- 了解了代码重用的价值。

### 11.4 复习测验

这些测验题将帮助你理解本章内容：

- 1) 本章所用的主 DirectSound 类的名称是什么？
- 2) 第二声音缓冲区是什么？
- 3) 在 DirectSound.h 中第二声音缓冲区的名称是什么？
- 4) 让声音循环播放所需的选项是什么？
- 5) 作为参考，绘制纹理（作为精灵）的函数的名称是什么？
- 6) 哪个 DXUT 助手类处理波形文件的装载？
- 7) 为了创建第二声音缓冲区，需要使用哪个 DXUT 助手函数？
- 8) 从用户的观点简要描述一下 DirectSound 处理声音混音的方法。
- 9) 由于 DirectMusic 已经过时了，在游戏中如果要回放音乐，有什么好的替代方法？
- 10) 在初始化 DirectSound 时要调用哪个函数？

### 11.5 自己动手

以下练习将帮助你跳出思维框框，跳开限制，增加理解能力。

习题 1 Play\_Sound 程序在球精灵每次击中某个缓冲器时播放音响效果。修改程序，让它按我们的选择显示不同数量的球，并且让每个球按随机的速度运动。

习题 2 Play\_Sound 程序在球精灵击中缓冲器时只播放一个声音。修改程序，多添加三个波形文件（要有相关代码来装载它们），在球击中缓冲器时随机播放其中的一种声音。

## 第 12 章 3D 渲染基础

本章讲解 3D 图形的基础知识。我们将学习基本的概念，以了解 3D 编程的关键所在。不过，本章不会对 3D 数学或图形理论有太详细的介绍，对于本书来说这些东西太过高级了。我们将学习实用的 3D 实现，以便编写简单的 3D 游戏。我们还将了解渲染简单的 3D 对象所需的知识，无需陷入理论之中。如果对矩阵数学的工作原理及 3D 渲染的工作原理有更低级别的问题，可以将本章当成起点，然后继续阅读更高级的书籍（见附录 B 中的推荐）。本章的目标只是介绍概念。

本章将学到：

- 如何创建并使用顶点。
- 如何操纵多边形。
- 如何创建带纹理的多边形。
- 如何创建立方体并旋转它。

### 12.1 3D 编程介绍

如今，每个人都有一块 3D 加速的视频卡，这早已是定论了。要不是市场的竞争每年都推动着越来越多的多边形的生成和新功能的出现，即使是低端的经济型视频卡装备的 3D 图形处理单元（GPU）也很吸引人。图 12-1 显示了一块典型的 3D 加速视频卡。

**建议** 本章讨论的 3D 编程，特指的是 Direct3D。这里讨论的有些 3D 概念只与 Direct3D 相关，而未必与任何其他图形 API 有关（也不是通常意义上的 3D 图形理论）。

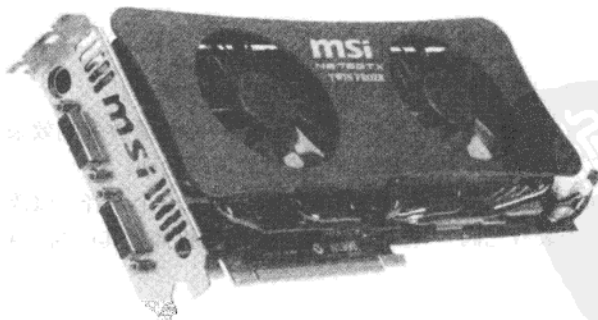


图 12-1 现代的 3D 视频卡具有生成实时的照片级真实感图形的能力  
(感谢 Micro Star International 提供图片)

#### 12.1.1 3D 编程的关键组成部分

在 Direct3D 中渲染一个场景，有三个关键组成部分：

1) **世界变换**。在“世界”中移动 3D 对象。“世界”这个词描述的是整个场景。换句话说,世界变化导致场景中的物体移动、旋转及缩放(一次一个对象)。

2) **视图变换**。打个比方说,这是照相机,它定义用户在屏幕上所看到的内容。这个照相机可以放置在“世界”中的任何地方。如果想移动照相机,只需进行视图变换即可。

3) **投影变换**。这是最后一步,在这里将视图变换的结果(照相机看到的对象)绘制到屏幕上,形成由像素组成的平面 2D 图像。投影确定已渲染的场景在屏幕上的样子(无论定义的屏幕纵横比如何影响输出)。

Direct3D 提供我们创建、渲染及查看场景所需的所有的函数和变换,无需使用任何 3D 数学知识。这对我们程序员来说再好不过了,因为 3D 矩阵数学不简单(假使我们自己从头编写矩阵数学计算的代码,它们有可能不如 Microsoft 的实现快)。

“变换”发生在我们对两个矩阵进行加、减、乘或除运算时,它导致结果矩阵的改变。这些改变导致 3D 对象移动、旋转及缩放。矩阵是一个大小为  $4 \times 4$  (或者 16 个单元格)的栅格或者二维数组。Direct3D 定义了我们在 3D 游戏中所需的所有标准矩阵。

### 12.1.2 3D 场景

我们必须先创建一个能够代表场景的 3D 对象,然后才能对其进行操作。本章将介绍从头开始创建简单的 3D 对象的方法,也会介绍 Direct3D 的一些主要以测试为目的而提供的免费模型。系统中有一些标准对象,例如圆柱体、金字塔、面包圈,甚至还有一个茶壶,可用于创建一个场景。

当然,创建整个 3D 游戏不能只使用源代码,因为在一个典型的游戏中有太多的对象了。我们最终会需要使用诸如 3ds max 或者免费的 Anim8or 建模程序(包括在 CD-ROM 中)来构建 3D 对象模型。下面两章将讲解如何将文件中的 3D 模型装载到场景中。不过在本章,重点是可编程的 3D 对象。

**建议** 本章和第 13 章只包含使用 Direct3D 进行 3D 渲染的基本知识,这足够用来渲染从 .X 文件中装载的、带有环境光照效果的带纹理的网格(mesh)了。我们不准在这本入门书籍中学习光照或着色器或者网格动画。如果你想继续学习更多使用 Direct3D 进行 3D 图形编程的方法,可参考附录 B 中推荐的书籍。

#### 1. 顶点

为视频卡增加性能的高级 3D 图形芯片看到的只有顶点。一个顶点是 3D 空间中的一个点,以 X、Y 和 Z 值来指定。视频卡本身实际上只“看”形成每个三角形的三个角的顶点。填充由这三个顶点所形成的空三角形是视频卡的工作。见图 12-2。与这个示例不同的是,在 3D 场景中所有的三角形都将是对其更为高效地进行渲染的直角三角形。

创建并操纵场景中的 3D 对象是我们程序员的工作,

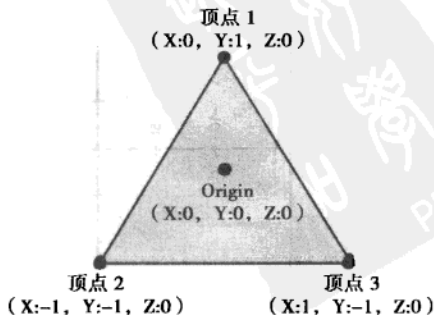


图 12-2 3D 场景完全由三角形组成



这有助于理解 3D 环境的一些基础知识。可以将整个场景当成带有三个轴的数学栅格。你如果学过几何或三角数学可能熟悉笛卡尔坐标系。这个坐标系是所有几何数学和三角数学的基础，在笛卡尔栅格中有许多公式和函数用来操纵其中的点。

## 2. 笛卡尔坐标系

这个“栅格”实际上是由两条相交于远点的无限长线组成的。这两条直线互相垂直。水平线称为 X 轴而垂直线称为 Y 轴。原点位于 (0,0)。X 轴越往右数值越大，越往左数值越小。同样的，Y 轴越往上数值越大，越往下数值越小。见图 12-3。

如果在笛卡尔坐标系中的某个位置上有一个点，例如 (100, -50)，那么可以使用数学计算来操纵这个点。可以对一个点做三种事情：

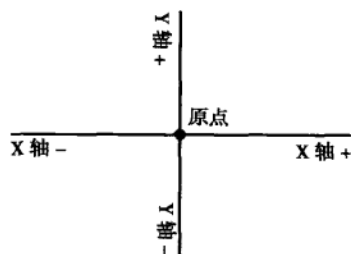


图 12-3 笛卡尔坐标系

- 1) **平移**。即将点移到新位置。见图 12-4。
- 2) **旋转**。即以原点为圆心以当前位置为半径旋转移动点。见图 12-5。
- 3) **缩放**。通过修改两个轴的整个范围，可以调整点与原点之间的相对位置。见图 12-6。

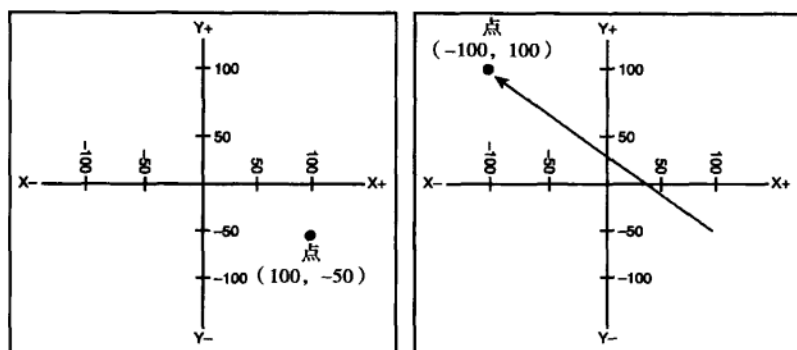


图 12-4 点 (100, -50) 平移 (-200, 150) 的结果是 (-100, 100)

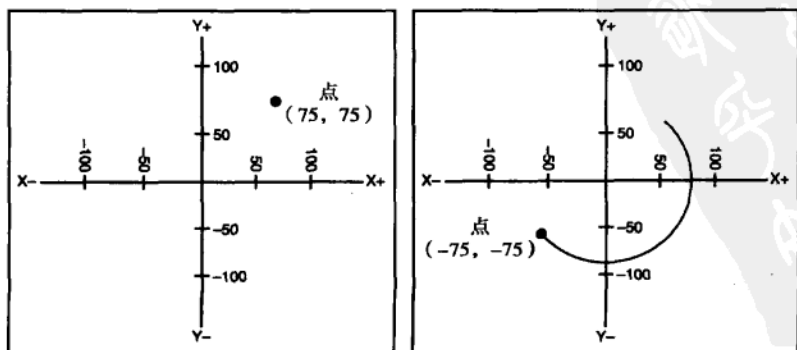


图 12-5 点 (75, 75) 旋转 180° 的结果是 (-75, -75)

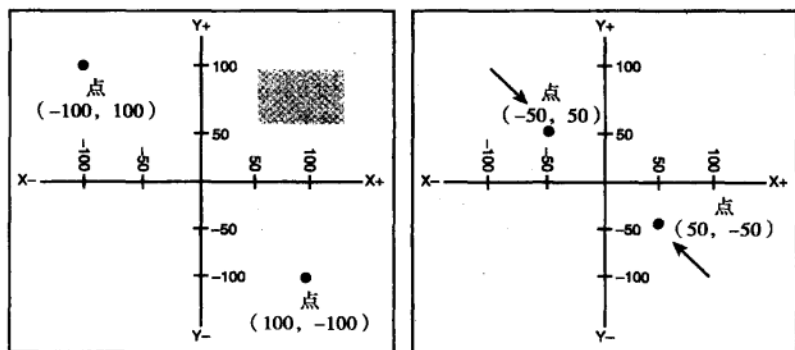


图 12-6 点  $(-100, 100)$  缩小 50% 的结果是  $(-50, 50)$

### 3. 顶点的原点

需要记住的是，在处理 3D 图形时，所有的一切都围绕着原点来工作。所以，如果想在屏幕上旋转一个 3D 对象，就必须记得所有的旋转都是以原点为基础的。如果将对象平移到新的不以原点为中心的位置，那么再对其旋转就会导致它以圆形绕着原点移动！

那么，这个问题怎么解决呢？这是大多数进入 3D 编程世界的人最大的症结所在，因为如果没有资历更高的程序员为你说清楚的话，要处理它实在是很难。通过本例，你将有机会学习 3D 图形编程重要却太经常被忽略的一课：技巧在于根本就不真的去移动 3D 对象。

什么？不，我不是在开玩笑。这个技巧是将所有的 3D 对象留在原点上并且根本不移动它们。这是不是把你搞晕了？好吧，我来解释。大家知道，3D 对象是由顶点组成的（确切地说每个三角形三个顶点）。关键的是，在指定位置上以指定的旋转和缩放值绘制 3D 对象，而不移动“原来”的对象本身。我的意思不是说要做一个它的副本，而是在刷新屏幕之前就在其最后的实例上绘制它。还记得在只有一个精灵图像时如何在屏幕上绘制许多精灵吗？移动 3D 对象与之有类似的地方，不同的只是以原来的“图像”为基础绘制 3D 对象，也就是说，原图像没有改变。将原对象留在原点上，我们就可以让各个对象围绕一个称为本地原点的点进行旋转，这样就保留了各个对象。

那么，如何不移动一个 3D 对象就实现移动效果呢？答案是使用矩阵。矩阵是一个  $4 \times 4$  的数字栅格，表示“空间”中的一个 3D 对象。场景（或游戏）中的每个 3D 对象都有其自己的矩阵。

**建议** 矩阵数学超出了本书的介绍范围，但如果想知道在多边形的世界里实际发生了什么，还需要深入学习更高级的 Direct3D 书籍。以笔者来看，在这个主题上讲得最好的是 Carl Granberg，他写了两本非常好的书：《Programming An RTS Game In Direct3D》和《Character Animation in Direct3D》（这两本书都由 Charles River 出版）。我在我的高级 DirectX 课程中使用这些书。

使用矩阵给每个 3D 对象指定它们自己的原点，这样我们的 3D 世界就有了自己的坐标系统，而场景中的所有对象也是如此，所以，我们可以彼此独立地操纵这些对象。甚至可以在不影响这

些独立对象的前提下操纵整个场景。例如，假设我们开发一款竞速游戏，汽车在椭圆赛道上赛跑。我们希望每辆汽车都尽可能真实，这就要求每辆车都能自己旋转和移动，无论其他车辆正在做什么。当然，在某些时候，我们还想加入一些代码让汽车在碰撞的时候撞毁。我们也想让汽车在赛道的路面上能够保持“平坦”，也就是说要计算赛道的角度并且适当地定位汽车的四个角。

把思维更推进一步，想一想如果一个对象还能包含子对象，每个子对象都有其自己的本地原点，它们遵照“父”对象的动作而动作时，那么会有哪些可能性出现。我们可以相对于父对象的原点对子对象定位，并让子对象自己旋转。这是不是有助于我们构思如何对汽车的轮子编程，让轮子在自己滚动的同时汽车保持不动呢？轮子“跟随”着汽车，也就是说它们同父对象一起平移/旋转/缩放，但它们有滚动及左转或右转的能力。

#### 4. 认真注意照相机

在编写了我们认为是干净的、“应该工作”的代码之后，却不能在屏幕上看到任何东西，这是3D编程最让人沮丧的问题了。在3D编程最常见的错误中排名第一的是忘记了照相机和视图转换。在阅读本章时，将以下要点牢记在心。

在场景中首先要做的是，使用一个测试的多边形或四边形来设置透视、照相机和视图，以便在继续之前确认场景已经正确设置。确认视图正确设置后，再继续为游戏编写代码。另外一个常见的问题涉及照相机的位置，在初始的测试中它可能一切正常，但后来可能显得过于靠近对象，对象也可能被“移出屏幕”。有一个好的测试方法是將照相机从原点移开（例如Z为-100），然后确认目标矩阵指向原点(0,0,0)。这样就可消除任何视图上的问题，从而全速回到游戏的编程上。

最初设置场景要做的第二件事情是检查场景的光照条件。是否启用了光照效果但却没有任何灯光？Direct3D实在是属于学术型的，除非你告诉它这里没有光源，否则它是不会主动创建环境光线的！

#### 12.1.3 转移到第三维

希望大家现在已经摸到笛卡尔坐标系的窍门了。虽然这对3D图形的学习至关重要，但我们不会深入任何更多细节，因为这一主题需要更多理论和讲解，这里没有这么多篇幅。我们仅讲解足够编写几个简单的3D游戏所需的内容，而后你可以自己决定要进一步学习3D编程的哪些方面。先写出能工作的代码并且编写实际的游戏，要比一次性地把像Direct3D这样的库的边边角角都学到家有趣得多。

**建议** 我们将在本章学习如何通过非常低级的、带有顶点缓冲区的图形编程来渲染简单的带纹理的立方体。在以这种方式处理顶点时，我们会深入Direct3D的内部核心，就如碰触GPU（图形处理单元）一样。在完成了对Direct3D低级别的探究之后，我们将专注于装载并绘制来自.X文件中的3D网格，这更容易一些。如果不想以这种方式来学习与Direct3D渲染管线有关的核心内容，可以跳到下一章（不过我强烈建议你阅读下来！）。

图12-7显示了笛卡尔坐标系中增加的第三维。所有和2D坐标系有关的规则都适用于3D，

但现在使用三个值 (X,Y,Z) 而不是两个值来引用一个点。

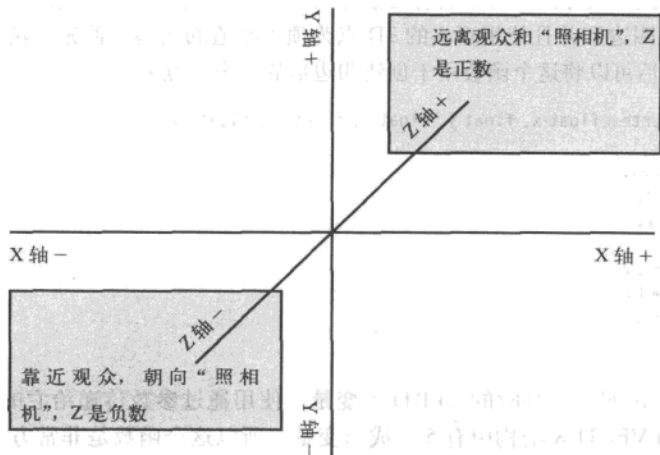


图 12-7 带有第三维的笛卡尔坐标系

#### 12.1.4 掌握 3D 管线

在屏幕上绘制单个多边形之前首先需要知道的是，Direct3D 使用由用户自定义的顶点格式。以下是我们将在本章使用的结构：

```
struct VERTEX
{
    float x, y, z;
    float tu, tv;
};
```

前三个成员变量是顶点的位置，而 *tu* 和 *tv* 变量用于表述纹理的绘制方法。现在我们对渲染过程的发生有难以置信的控制能力了。这两个变量告诉 Direct3D 在表面上绘制纹理的方法，Direct3D 支持纹理围绕 3D 对象曲线进行包裹。我们使用 *tu* = 0.0、*tv* = 0.0 来指定纹理的左上角；使用 *tu* = 1.0、*tv* = 1.0 来指定纹理的右下角。这两者之间的所有多边形的纹理坐标通常是零，以此告诉 Direct3D 就在它们上面伸展纹理。

纹理是比较高级的内容，随着对 3D 编程的深入，你将发现它有成百上千个选项。目前，我们只在一个四边形中的两个三角形上伸展纹理。

#### 四边形

以 VERTEX 结构为基础，我们可以创建出一个能够帮助我们创建并保存四边形的结构：

```
struct QUAD
{
    VERTEX vertices[4];
    LPDIRECT3DVERTEXBUFFER9 buffer;
    LPDIRECT3DTEXTURE9 texture;
};
```

QUAD 结构完全是个自给自足的结构。它有四边形（由两个三角形组成）的四个角落的四个顶点；有四边形的顶点缓冲区（马上会有更多介绍）；还有映射到两个三角形上的纹理。很酷吧？唯一缺少的是创建四边形并且使用真实的 3D 点为顶点赋值的代码。首先，我们编写一个创建单个顶点的函数。而后可以将这个函数用于创建四边形的四个顶点：

```
VERTEX CreateVertex(float x, float y, float z, float tu, float tv)
{
    VERTEX vertex;
    vertex.x = x;
    vertex.y = y;
    vertex.z = z;
    vertex.tu = tu;
    vertex.tv = tv;
    return vertex;
}
```

这个函数只是声明一个临时的 VERTEX 变量，使用通过参数传递给它的值赋值这个变量，然后返回它。因为 VERTEX 结构中有 5 个成员变量，所以这个函数是非常方便的。稍后将介绍如何创建及绘制四边形的方法，不过需要先学习与顶点缓冲区有关的知识。

### 12.1.5 顶点缓冲区

顶点缓冲区实际上没那么可怕。我对顶点缓冲区的第一印象是它是某种类型的表面，3D 对象就绘制在它上面，然后送往屏幕，就如 3D 的双缓冲区。这其实错得太离谱了！顶点缓冲区其实只是我们储存组成多边形的点的地方，从而 Direct3D 可以绘制多边形。在程序中，只要需要，从技术上说可以有許多顶点缓冲区——每个三角形一个。

为了能够清楚地描述，安排每个对象都有其自己的顶点缓冲区，然后就此来介绍获得屏幕上的 3D 对象的方法。由于本章内容是以四边形（由两个排成一“条”的三角形组成）的概念为基础的，所以为场景中的每个四边形都创建一个顶点缓冲区，以帮助大家理解这里的内容。而且，如果你是第一次学习这些内容，那么它的确有所帮助。让每个四边形都有一个顶点缓冲区可以清楚地说明渲染四边形时所发生的一切。

从优化和效率方面来说，顶点缓冲区是一个需要大量讨论的问题。实际上，大多数 3D 引擎都采用所谓的顶点缓冲区缓存，它包含在照相机的视图中所有可见的顶点。强大的 3D 引擎也使用所谓的纹理缓存，以便共享同一纹理的多边形重用它们。要是你对为什么要这样做感到好奇，只要理解 3D 卡一次只能“使用”一个纹理就行了。所以，告诉 Direct3D 一次只使用一个纹理，然后在场景中任何需要这一纹理的多边形上使用这一纹理，接下来再处理下一个纹理，这样做将会更有效率。这就是纹理缓存的用途，它会处理所有这些问题。

#### 1. 创建顶点缓冲区

首先必须为顶点缓冲区定义一个变量：

```
LPDIRECT3DVERTEXBUFFER9 buffer;
```

接下来，可通过使用 CreateVertexBuffer 函数来创建顶点缓冲区。其格式如下：

```

HRESULT CreateVertexBuffer(
    UINT Length,
    DWORD Usage,
    DWORD FVF,
    D3DPool Pool,
    IDirect3DVertexBuffer9** ppVertexBuffer,
    HANDLE* pSharedHandle
);

```

第一个参数指定顶点缓冲区的尺寸，它必须足够大，以便保存想要渲染的多边形的所有顶点。第二个参数指定访问顶点缓冲区的方法，通常是只写。第三个参数指定 Direct3D 将要接收的顶点流的类型。应该将与我们所创建的顶点结构类型相关的值传递给它。这里的每个顶点只有位置和纹理坐标，所以这个值将是 D3DFVF\_XYZ | D3DFVF\_TEX1（注意，这个值是以或运算组合的）。以下是定义顶点格式的代码：

```
#define D3DFVF_MYVERTEX (D3DFVF_XYZ | D3DFVF_TEX1)
```

第五个参数指定顶点缓冲区指针，最后一个参数不需要。下面给出了一个示例：

```

d3ddev->CreateVertexBuffer(
    4*sizeof(VERTEX),
    D3DUSAGE_WRITEONLY,
    D3DFVF_MYVERTEX,
    D3DPool_DEFAULT,
    &buffer,
    NULL);

```

不难看出，第一个参数接收一个 sizeof(VERTEX) 乘以 4 的整数（因为一个四边形有 4 个顶点）。如果只绘制一个三角形，那么应该指定的值是 3 \* sizeof(VERTEX)，3D 对象有多少个顶点就乘以多少。而唯有顶点缓冲区长度和指针（分别是第一和第五个参数）是真正重要的参数。

## 2. 填写顶点缓冲区

创建顶点缓冲区的最后一个步骤是使用多边形的实际顶点来填写它。这个步骤必须跟随在任何生成或装载顶点数组的代码之后，因为它会将数据放入到顶点缓冲区中。为了方便参考，以下再次给出 QUAD 结构的定义（尤其注意 VERTEX 数组）：

```

struct QUAD
{
    VERTEX vertices[4];
    LPDIRECT3DVERTEXBUFFER9 buffer;
    LPDIRECT3DTEXTURE9 texture;
};

```

举个例子来说，可使用 CreateVertex 函数为一个四边形设置默认值：

```

vertices[0] = CreateVertex(-1.0f, 1.0f, 0.0f, 0.0f, 0.0f);
vertices[1] = CreateVertex(1.0f, 1.0f, 0.0f, 1.0f, 0.0f);
vertices[2] = CreateVertex(-1.0f, -1.0f, 0.0f, 0.0f, 1.0f);
vertices[3] = CreateVertex(1.0f, -1.0f, 0.0f, 1.0f, 1.0f);

```

这只是将数据填入顶点的一种方式。可以在程序的某个地方定义不同类型的多边形或者从文件装载 3D 形状（第 13 章对此有更多介绍）。

在有了顶点数据之后，可以将其放入顶点缓冲区中。在放的过程中必须锁住（Lock）顶点缓冲区，并将顶点复制到顶点缓冲区中，然后对顶点缓冲区解锁（Unlock）。这需要一个临时指针。以下是设置用于 Direct3D 的带有数据的顶点缓冲区的方法：

```

void *temp = NULL;
buffer->Lock( 0, sizeof(vertices), (void**)&temp, 0);
memcpy(temp, vertices, sizeof(vertices));
buffer->Unlock();

```

作为参考，以下是 Lock 的定义。第二个和第三个参数是重要参数，一个指定缓冲区的长度，另一个是指向缓冲区的指针。

```

HRESULT Lock(
    UINT OffsetToLock,
    UINT SizeToLock,
    VOID **ppbData,
    DWORD Flags
);

```

### 12.1.6 渲染顶点缓冲区

在初始化顶点缓冲区之后，就为 Direct3D 图形管线做好了准备，源顶点就不再有用了。这称为“设置”，而且是最近几年被移出 Direct3D 驱动程序，转而由 GPU 来实现的功能之一。让硬编码的芯片从顶点缓冲区中将顶点和纹理放到场景中要比软件快得多。

最后，所有的一切就只与渲染顶点缓冲区内的内容有关了，我们下面学习如何实现。要将当前正在处理的顶点缓冲区发送到屏幕上，需要设置 Direct3D 设备的流源（stream source），让它指向顶点缓冲区，然后调用 DrawPrimitive 函数。在此之前，必须首先设置要使用的纹理。这是 3D 图形最难以理解的地方，尤其对初学者。Direct3D 一次只处理一个纹理，所以我们每次更换纹理时必须告诉 Direct3D，否则它将在整个场景中一直使用最后那次定义的纹理！有点奇怪？呃，如果仔细想想就会发现这其实是合理的。没有预编程的方法可以告诉 Direct3D 给某个多边形使用“这个”纹理而给另一个多边形使用“那个”纹理。每次需要更改纹理时我们必须自己编写代码。

在四边形的情况中，每个四边形我们只处理一个单一的纹理，因而这个概念更容易掌握一些。可以创建任何尺寸的顶点缓冲区，但通过让每个四边形都有其自己的顶点缓冲区，可以使我们更容易理解 3D 渲染的工作原理。这不是最高效的在屏幕上绘制 3D 对象的方法，但如果是在学习基础知识，那么这是极好的方式！每个四边形有一个顶点缓冲区和一个纹理，于是渲染四边形的源代码就清楚明了。不难想象，由于我们可以编写一个函数来绘制一个顶点缓冲区和纹理都

可在 QUAD 结构中很容易访问的四边形，事情就容易得多了。

首先为四边形设置纹理：

```
d3ddev->SetTexture(0, texture);
```

接下来，设置流源，以便 Direct3D 知道顶点的来源及需要渲染的有多少：

```
d3ddev->SetStreamSource(0, q.buffer, 0, sizeof(VERTEX));
```

最后，绘制流源指定的基元（primitive），包括渲染方法、起始顶点及要绘制的多边形数量：

```
d3ddev->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
```

显然，这三个函数可一起放入一个可重用的 Draw 函数中（稍后会介绍更多相关内容）。

### 12.1.7 创建四边形

我不知道四边形是否是一个正式的术语，但这没什么关系，因为这个术语从两个方面描述了我想要做的事情。四边形这个术语的一个方面是，它表示矩形的四个角，而矩形是大多数 3D 场景的构建块。我们可以使用一堆立方体（每个立方体由六个四边形组成）来构建几乎任何东西。不难猜测，这些角都是以顶点来表示的。四边形的另一个方面是，它表示了一个三角形条（triangle strip）的四个顶点。

#### 1. 绘制三角形

绘制对象（对象都由三角形组成）有两种方法：

- 独立绘制每一个多边形的三角形列表，每个由三个一组的顶点组成。
- 一个三角形条绘制许多通过共享顶点连接在一起的多个多边形。

显然，第二种方法更高效，所以更为可取，而且它能帮助提高渲染速度，因为它所用的顶点更少。不过我们不能使用三角形条来渲染整个场景，因为大多数对象并不是互相连接的。目前，三角形条非常适用于地面地形、建筑物和其他大对象。对于诸如游戏中的角色这样的小对象，它也能良好工作。不过，无论所有的三角形是位于单个顶点缓冲区中还是多个顶点缓冲区中，Direct3D 都将以同样的速度渲染场景，理解这一特性在这里是有帮助的。把它想象成一系列的 for 循环。告诉我这两段代码哪一段更快。忽略 num++ 这一部分，只假设在循环中“发生着一些有用的事情”。

```
for (int n=0; n<1000; n++) num++;
```

还是：

```
(for int n=0; n<250; n++) num++;  
(for int n=0; n<250; n++) num++;  
(for int n=0; n<250; n++) num++;  
(for int n=0; n<250; n++) num++;
```



你是怎么想的呢？似乎第一段代码更快是显而易见的，因为调用更少。而对于深入考虑优化的人可能觉得第二段代码更快，因为也许它避免了这里那里的 if 语句（将循环展开并将 if 语句放到循环外面总是更快）。

**展开循环** 展开循环是什么意思呢（这与 3D 没有直接关系，但肯定有帮助）？看一看以下两组代码（来自一个虚构的画线函数。假设  $x$  和  $y$  已经定义）：

```
for (x=0; x<639; x++)
    if (x%2 == 0)
        DrawPixel(x, y, BLUE);
    else
        DrawPixel(x, y, RED);
and
for (x=0; x<639; x+=2)
    DrawPixel(x, y, BLUE);
for (x=1; x<639; x+=2)
    DrawPixel(x, y, RED);
```

第二个代码片段可能要比第一段快一倍，因为循环被展开了，if 语句被移除了。在编写游戏时要考虑这样的优化问题，因为我们必须仔细编写循环代码。for 循环本身不实际占用任何处理器时间，需要重点考虑的是 for 循环执行的代码。

一个四边形由两个三角形组成。四边形只需要四个顶点，因为两个三角形会以三角形条来绘制。两种类型的三角形渲染方法之间的区别见图 12-8。

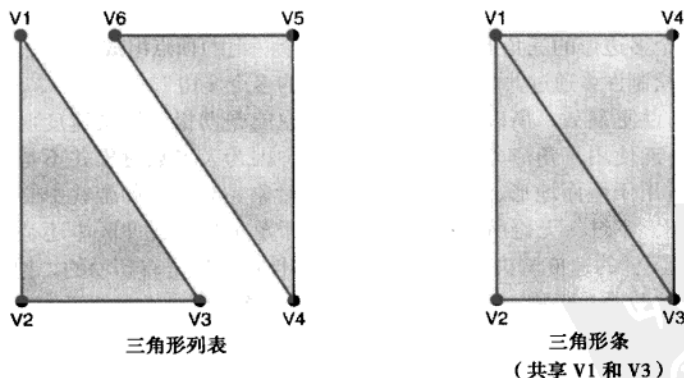


图 12-8 三角形列表和三角形条渲染方法的比较

图 12-9 显示了三角形条的其他可能的形式。任何共享一个边的两个顶点都可以连接起来

## 2. 创建四边形

创建四边形所需的工作甚至要比创建两个互相连接的三角形更少，这要归功于三角形条渲染过程。绘制任何多边形，无论是三角形、四边形还是复杂形状，都涉及两个基本步骤。

首先，必须将顶点复制到 Direct3D 顶点流中。做这件事首先必须锁住顶点缓冲区，然后将

顶点复制到由一个指针变量指向的临时存储位置，然后对顶点缓冲区解锁。

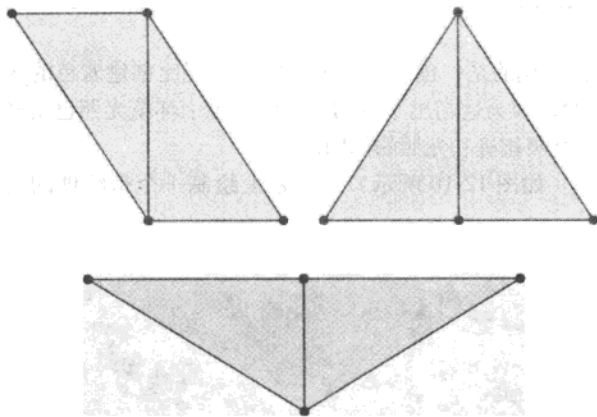


图 12-9 三角形条可以有多种形式。还要注意的，可以使用的多边形要比两个多得多

```
void *temp = NULL;
quad->buffer->Lock(0, sizeof(quad->vertices), (void**)&temp, 0);
memcpy(temp, quad->vertices, sizeof(quad->vertices));
quad->buffer->Unlock();
```

下一步是设置纹理，告诉 Direct3D 在哪里找到包含顶点的流源，然后调用 DrawPrimitive 函数绘制在顶点缓冲区流中指定的多边形。我乐于将这当成 Star Trek（星际迷航）式的运输工具。多边形从顶点缓冲区中被运输到了流中，然后在屏幕上重新组装！

```
d3ddev->SetTexture(0, quad->texture);
d3ddev->SetStreamSource(0, quad->buffer, 0, sizeof(VERTEX));
d3ddev->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
```

**建议** 现在我们已经对使用 Direct3D 中非常低级的 3D 图形编程功能（也就是通过操作和渲染多边形顶点）来创建及渲染各个由顶点组成的多边形有了理解，并且知道如何让其工作，可以说，你已经对 3D 渲染的内部工作上手了。你应该感觉良好才对！不是每个人都有能力掌握 3D 图形编程的，更不用说精通它并且制作高质量的场景了。

好在过了这一章我们就不再使用顶点缓冲区来处理问题了。下面将带领读者一起体验一个复杂的示例（很快就会讲到的 Textured Cube Demo），然后我们将继续前进，学习装载和渲染使用 .X 文件格式的网格文件。

## 12.2 带纹理的立方体示例

我们来做点实际的东西吧。没有人会在意绘制着色的三角形，所以我们不在这一主题上浪费时间。你是否准备通过对三角形编程来将三角形组装到对象中，然后移动它们并且进行碰撞检查

等工作，从而创建一个完整的 3D 游戏？当然不是，所以我们为什么要花时间学习它？对于 3D 系统而言三角形是至关重要的，但如果只有一个三角形则没什么用处。只有将三角形组合起来，才使事情变得有趣。

现代 3D API 中真正有趣的是，创建带纹理的四边形要比创建着色的四边形更容易。这里将避免提及动态光照的主题，因为这超出了本书的介绍范围；环境光照已足够我们使用了。我们将在第 13 章中更详细地探究照相机和光照的问题。

Textured\_Cube 程序（如图 12-10 所示）在屏幕上绘制一个带纹理的立方体并在 x 和 z 轴上旋转它。

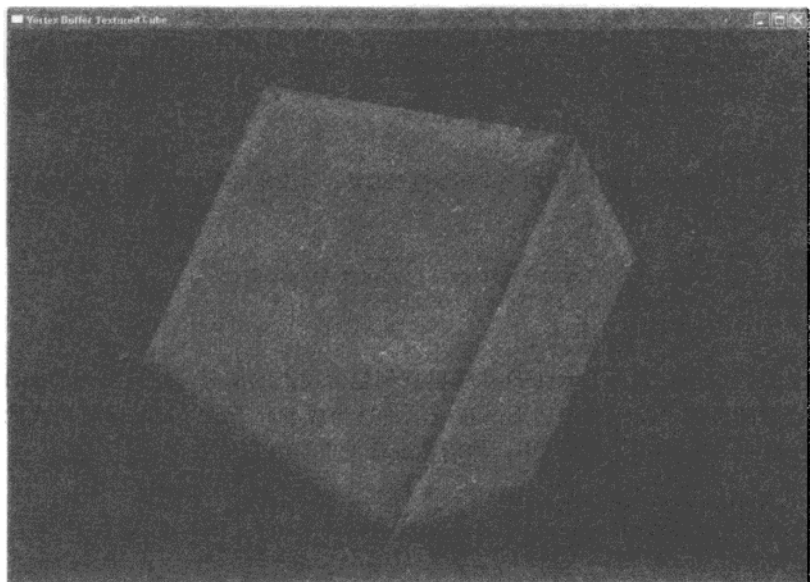


图 12-10 Textured\_Cube 程序演示渲染一个基于顶点缓冲区的从头设计的立方体的方法

虽然似乎在一个立方体内只有 8 个顶点（见图 12-11），但实际上要多得多，因为每个三角形必须有由其自己的三个顶点组成的集合。不过我们很快就会了解，三角形条可以很好地用四个顶点来生成一个四边形。

有了三角形和四边形的基础，现在可以简要介绍一下立方体了。立方体被认为是我们可以创建的最简单的 3D 对象，也是很好的示例形状，因为它有 6 个相等的面。就如所有 3D 环境中的对象都必须由三角形组成一样，立方体也必须由三角形组成。实际上，立方体的每个面（矩形）是把两组直角边以相对的方向将两个三角形并排放置形成的结果。见图 12-12。

**建议** 直角三角形是有一个  $90^\circ$  角的三角形，它是 3D 图形的首选形状。如果不为视频卡提供直角三角形，那么它会把奇形怪状的三角形分解为两个或更多直角三角形——是的，就是这么重要！

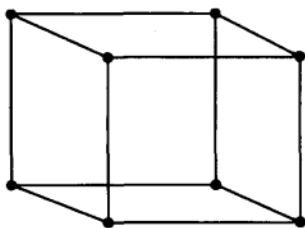


图 12-11 一个立方体有 8 个角，每个都由一个顶点来表示

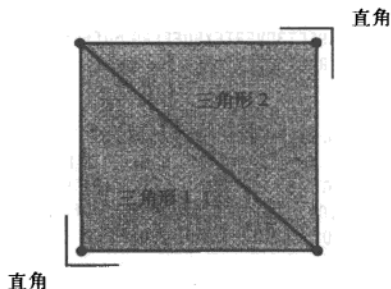


图 12-12 一个矩形是由两个直角三角形组成的

在使用三角形组装出一个立方体之后，就会得到如图 12-13 这样的东西。这张图展示了分解成许多三角形的立方体。

### MyGame.cpp

下面给出 Textured\_Cube 程序的源代码。建议读者从 CD-ROM 装载项目（为了方便使用可复制到硬盘上）。如果觉得用如下所示的代码来创建一个 3D 模型显得很奇怪的话，那就对了。这的确很突兀，但在目前，为了说明如何使用顶点来构建多边形进而建模（也称为网格），它还是有帮助的。用这样的代码来创建任何类型的复杂 3D 模型都将非常困难，所以它仅适用于这么一个简单的示例。我们很快将学习如何将网格文件装载到内存中，然后渲染一个带有光照效果的复杂网格。

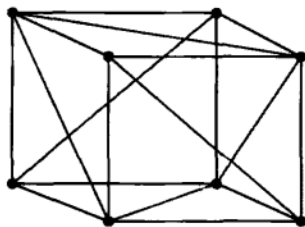


图 12-13 立方体由 6 个面组成，一共有 12 个三角形

**建议** 下面给出了 Textured\_Cube 程序的完整源代码。这时无需对任何支持文件进行更改。

```
#include "MyDirectX.h"
using namespace std;

const string APPTITLE = "Vertex Buffer Textured Cube";
const int SCREENW = 1024;
const int SCREENH = 768;

DWORD screentimer = timeGetTime();

//vertex and quad definitions
#define D3DFVF_MYVERTEX (D3DFVF_XYZ | D3DFVF_TEX1)
struct VERTEX
{
    float x, y, z;
    float tu, tv;
};
struct QUAD
{

```



```

    VERTEX vertices[4];
    LPDIRECT3DVERTEXBUFFER9 buffer;
    LPDIRECT3DTEXTURE9 texture;
};

VERTEX cube[] = {
    {-1.0f, 1.0f, -1.0f, 0.0f, 0.0f}, //side 1
    { 1.0f, 1.0f, -1.0f, 1.0f, 0.0f },
    {-1.0f, -1.0f, -1.0f, 0.0f, 1.0f },
    { 1.0f, -1.0f, -1.0f, 1.0f, 1.0f },

    {-1.0f, 1.0f, 1.0f, 1.0f, 0.0f }, //side 2
    {-1.0f, -1.0f, 1.0f, 1.0f, 1.0f },
    { 1.0f, 1.0f, 1.0f, 0.0f, 0.0f },
    { 1.0f, -1.0f, 1.0f, 0.0f, 1.0f },

    {-1.0f, 1.0f, 1.0f, 0.0f, 0.0f }, //side 3
    { 1.0f, 1.0f, 1.0f, 1.0f, 0.0f },
    {-1.0f, 1.0f, -1.0f, 0.0f, 1.0f },
    { 1.0f, 1.0f, -1.0f, 1.0f, 1.0f },

    {-1.0f, -1.0f, 1.0f, 0.0f, 0.0f }, //side 4
    {-1.0f, -1.0f, -1.0f, 1.0f, 0.0f },
    { 1.0f, -1.0f, 1.0f, 0.0f, 1.0f },
    { 1.0f, -1.0f, -1.0f, 1.0f, 1.0f },

    { 1.0f, 1.0f, -1.0f, 0.0f, 0.0f }, //side 5
    { 1.0f, 1.0f, 1.0f, 1.0f, 0.0f },
    { 1.0f, -1.0f, -1.0f, 0.0f, 1.0f },
    { 1.0f, -1.0f, 1.0f, 1.0f, 1.0f },

    {-1.0f, 1.0f, -1.0f, 1.0f, 0.0f }, //side 6
    {-1.0f, -1.0f, -1.0f, 1.0f, 1.0f },
    {-1.0f, 1.0f, 1.0f, 0.0f, 0.0f },
    {-1.0f, -1.0f, 1.0f, 0.0f, 1.0f }
};

QUAD *quads[6];
D3DXVECTOR3 cameraSource;
D3DXVECTOR3 cameraTarget;

void SetPosition(QUAD *quad, int ivert, float x, float y, float z)
{
    quad->vertices[ivert].x = x;
    quad->vertices[ivert].y = y;
    quad->vertices[ivert].z = z;
}

void SetVertex(QUAD *quad, int ivert, float x, float y, float z, float tu, float tv)
{
    SetPosition(quad, ivert, x, y, z);
    quad->vertices[ivert].tu = tu;
    quad->vertices[ivert].tv = tv;
}

```

```
VERTEX CreateVertex(float x, float y, float z, float tu, float tv)
{
    VERTEX vertex;
    vertex.x = x;
    vertex.y = y;
    vertex.z = z;
    vertex.tu = tu;
    vertex.tv = tv;
    return vertex;
}

QUAD *CreateQuad(char *textureFilename)
{
    QUAD *quad = (QUAD*)malloc(sizeof(QUAD));

    //load the texture
    D3DXCreateTextureFromFile(d3ddev, textureFilename, &quad->texture);
    //create the vertex buffer for this quad
    d3ddev->CreateVertexBuffer(
        4*sizeof(VERTEX),
        0,
        D3DFVF_MYVERTEX, D3DPool_DEFAULT,
        &quad->buffer,
        NULL);

    //create the four corners of this dual triangle strip
    //each vertex is X,Y,Z and the texture coordinates U,V
    quad->vertices[0] = CreateVertex(-1.0f, 1.0f, 0.0f, 0.0f, 0.0f);
    quad->vertices[1] = CreateVertex( 1.0f, 1.0f, 0.0f, 1.0f, 0.0f);
    quad->vertices[2] = CreateVertex(-1.0f, -1.0f, 0.0f, 0.0f, 1.0f);
    quad->vertices[3] = CreateVertex( 1.0f, -1.0f, 0.0f, 1.0f, 1.0f);

    return quad;
}

void DeleteQuad(QUAD *quad)
{
    if (quad == NULL)
        return;

    //free the vertex buffer
    if (quad->buffer != NULL)
        quad->buffer->Release();

    //free the texture
    if (quad->texture != NULL)
        quad->texture->Release();

    //free the quad
    free(quad);
}

void DrawQuad(QUAD *quad)
{

```

```

//fill vertex buffer with this quad's vertices
void *temp = NULL;
quad->buffer->Lock(0, sizeof(quad->vertices), (void**)&temp, 0);
memcpy(temp, quad->vertices, sizeof(quad->vertices));
quad->buffer->Unlock();
//draw the textured dual triangle strip
d3ddev->SetTexture(0, quad->texture);
d3ddev->SetStreamSource(0, quad->buffer, 0, sizeof(VERTEX));
d3ddev->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
}

void SetIdentity()
{
    //set default position, scale, and rotation
    D3DXMATRIX matWorld;
    D3DXMatrixTranslation(&matWorld, 0.0f, 0.0f, 0.0f);
    d3ddev->SetTransform(D3DTS_WORLD, &matWorld);
}

void ClearScene(D3DCOLOR color)
{
    d3ddev->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER, color, 1.0f, 0);
}

void SetCamera(float x, float y, float z, float lookx, float looky, float lookz)
{
    D3DXMATRIX matView;
    D3DXVECTOR3 updir(0.0f, 1.0f, 0.0f);

    //move the camera
    cameraSource.x = x;
    cameraSource.y = y;
    cameraSource.z = z;

    //point the camera
    cameraTarget.x = lookx;
    cameraTarget.y = looky;
    cameraTarget.z = lookz;

    //set up the camera view matrix
    D3DXMatrixLookAtLH(&matView, &cameraSource, &cameraTarget, &updir);
    d3ddev->SetTransform(D3DTS_VIEW, &matView);
}

void SetPerspective(float fieldOfView, float aspectRatio, float nearRange,
float farRange)
{
    //set the perspective so things in the distance will look smaller
    D3DXMATRIX matProj;
    D3DXMatrixPerspectiveFovLH(&matProj, fieldOfView, aspectRatio,
nearRange, farRange);
    d3ddev->SetTransform(D3DTS_PROJECTION, &matProj);
}

```

```
void init_cube()
{
    for (int q=0; q<6; q++)
    {
        int i = q*4; //little shortcut into cube array
        quads[q] = CreateQuad("cube.bmp");
        for (int v=0; v<4; v++)
        {
            quads[q]->vertices[v] = CreateVertex(
                cube[i].x, cube[i].y, cube[i].z,           //position
                cube[i].tu, cube[i].tv);                   //texture coords
            i++; //next vertex
        }
    }
}

bool Game_Init(HWND window)
{
    srand(time(NULL));

    //initialize Direct3D
    if (!Direct3D_Init(window, SCREENW, SCREENH, false))
    {
        MessageBox(window, "Error initializing Direct3D", APPTITLE.c_str(), 0);
        return false;
    }

    //initialize DirectInput
    if (!DirectInput_Init(window))
    {
        MessageBox(window, "Error initializing DirectInput",
            APPTITLE.c_str(), 0);
        return false;
    }

    //initialize DirectSound
    if (!DirectSound_Init(window))
    {
        MessageBox(window, "Error initializing DirectSound",
            APPTITLE.c_str(), 0);
        return false;
    }

    //position the camera
    SetCamera(0.0f, 2.0f, -3.0f, 0, 0, 0);

    float ratio = (float)SCREENW / (float)SCREENH;
    SetPerspective(45.0f, ratio, 0.1f, 10000.0f);

    //turn dynamic lighting off, z-buffering on
    d3ddev->SetRenderState(D3DRS_LIGHTING, FALSE);
    d3ddev->SetRenderState(D3DRS_ZENABLE, TRUE);
}
```



```
//set the Direct3D stream to use the custom vertex
d3ddev->SetFVF(D3DFVF_MYVERTEX);

//convert the cube values into quads
init_cube();

return true;
}

void rotate_cube()
{
    static float xrot = 0.0f;
    static float yrot = 0.0f;
    static float zrot = 0.0f;

    //rotate the x and y axes
    xrot += 0.05f;
    yrot += 0.05f;

    //create the matrices
    D3DXMATRIX matWorld;
    D3DXMATRIX matTrans;
    D3DXMATRIX matRot;
    //get an identity matrix
    D3DXMatrixTranslation(&matTrans, 0.0f, 0.0f, 0.0f);

    //rotate the cube
    D3DXMatrixRotationYawPitchRoll(&matRot,
                                   D3DXToRadian(xrot),
                                   D3DXToRadian(yrot),
                                   D3DXToRadian(zrot));
    matWorld = matRot * matTrans;

    //complete the operation
    d3ddev->SetTransform(D3DTS_WORLD, &matWorld);
}

void Game_Run(HWND window)
{
    if (!d3ddev) return;
    DirectInput_Update();
    d3ddev->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                D3DCOLOR_XRGB(0,0,100), 1.0f, 0);

    // slow rendering to approximately 60 fps
    if (timeGetTime() > screentimer + 14)
    {
        screentimer = GetTickCount();

        rotate_cube();

        //start rendering
```



```

    if (d3ddev->BeginScene())
    {
        for (int n=0; n<6; n++)
            DrawQuad(quads[n]);

        //stop rendering
        d3ddev->EndScene();
        d3ddev->Present(NULL, NULL, NULL, NULL);
    }

    //exit with escape key or controller Back button
    if (KEY_DOWN(VK_ESCAPE)) gameover = true;
    if (controllers[0].wButtons & XINPUT_GAMEPAD_BACK) gameover = true;
}

void Game_End()
{
    for (int q=0; q<6; q++)
        DeleteQuad(quads[q]);

    DirectSound_Shutdown();
    DirectInput_Shutdown();
    Direct3D_Shutdown();
}

```

### 12.3 你所学到的

本章介绍了 3D 图形编程的概况。我们学习了许多与 Direct3D 有关的知识，并且还做了一个带纹理的立方体演示。有以下几个要点：

- 什么是顶点及它们如何组成一个三角形。
- 如何创建一个顶点结构。
- 学习了与三角形条和三角形列表有关的知识。
- 如何创建顶点缓冲区并向其填入顶点。
- 学习了四边形及创建它们的方法。
- 纹理映射。
- 如何创建一个旋转立方体。

### 12.4 复习测验

以下测验将帮助读者巩固在本章所学的知识。

- 1) 什么是顶点？
- 2) 顶点缓冲区的作用是什么？
- 3) 在一个四边形中有多少顶点？
- 4) 一个四边形由几个三角形组成？
- 5) 绘制多边形的 Direct3D 函数的名称是什么？



- 6) 灵活的顶点缓冲区有什么作用?
- 7) 用于表示顶点 X,Y,Z 值的最常见的数据类型是什么?
- 8) 将角从角度转换为弧度的 DirectX 函数是什么?
- 9) 我们通常用于将大量顶点数据复制到顶点缓冲区中的 C 函数是什么?
- 10) 表示我们从虚拟照相机中所看到的内容的标准矩阵是哪一个?

## 12.5 自己动手

以下习题将考验读者对本章知识的记忆能力。

习题 1 Textured\_Cube 程序创建一个带有纹理的旋转立方体。修改这个程序, 让立方体根据键盘的输入旋转得更快或更慢。

习题 2 修改 Textured\_Cube 程序, 让立方体 6 个面中的每一面都有不同的纹理。提示: 可从 DrawQuad 函数中将代码复制到主源代码文件中, 以便使用不同的纹理。



## 第 13 章 渲染 3D 模型文件

本章是第 12 章的自然后续，在第 12 章我们学习了使用原始的顶点缓冲区从头创建及渲染带有纹理的 3D 立方体的方法。这是个很好的学习体验，但手工编写代码能做出来的网格也就只能是像立方体这样的东西了。本章要往前进一步，教授在运行时使用特殊的 Direct3D 函数（这个函数生成诸如立方体、球体和圆柱体这样的网格形状）来创建后援 3D 网格的方法，以及从 .X 文件中将网格装载到内存并使用纹理来渲染它的方法。本章我们将只使用环境光照。

本章将学到：

- 如何创建及渲染一个后援网格。
- 如何将网格文件装载到内存中。
- 如何变换及渲染网格。

### 13.1 创建及渲染后援网格

我发现把探究 Direct3D 的后援网格函数作为学习 3D 网格渲染的开始很有帮助。我们能够创建“运行时”网格，也就是说，不是从文件中装载的网格，而是在程序运行时用算法创建的。将它们称为后援网格是因为它们内建于 Direct3D 中而且可以在任何时候创建。实际上，在制作某些类型的需要快速生成对象的游戏时后援网格极为有用。例如，滚动发射器中的子弹。

**建议** 网格是由顶点组成的 3D 对象。“3D 模型”这个术语已被弃用。

#### 13.1.1 创建后援网格

Direct3D 有许多可以通过一个返回 ID3DXMesh 对象的函数来动态创建的后援网格对象。以下是我们可以在运行时使用 Direct3D 创建的一些网格：

- 立方体
- 球体
- 圆柱体
- 面包圈（圆环）
- 茶壶

在 Direct3D 中有分别用于创建这些后援网格的函数（还有一个用于创建 3D 文本和简单多边形的函数，这里没有介绍）。在调用这些函数时，我们将传递一个指向 ID3DXMesh 对象的指针，其定义方法如下：

```
LPD3DXMESH mesh;
```

注意，LPD3DXMESH 只是一个预定义的指向 ID3DXMesh 对象的指针，它在 Direct3D 中如

此定义：

```
#define LPD3DXMESH *ID3DXMesh;
```

在定义网格对象时我们既可使用 LPD3DXMESH 也可使用 \*ID3DXMesh，其结果是相同的。笔者倾向于使用前一种方法，因为它与命名规范更为一致。

**建议** 虽然直接光照和着色器要更为吸引人，但这些对象超出了本书介绍的范围，而且要想讲解它们将需要大量篇幅！因此，建议读者参考附录 B 中的书籍来进一步学习。此外，在 CD-ROM 中提供了许多光照和着色器的示例（见 \sources\AppendixD）以便你进一步学习，其显示效果在附录 D 中给出。这些示例也许不遵从本书使用 MyDirectX 文件的套路，但它们与本书共享着许多相同的函数。

### 1. 建立立方体

假设我们已经按照上面讲解的方法定义了网格对象，那么下面我们就可以使用 D3DXCreateBox 函数来建立立方体上的顶点。在本章的后面将介绍一个名为 Stock\_Mesh 的示例程序，演示使用这些函数创建及渲染后援网格的方法。图 13-1 显示了该程序的这一版本的输出。

```
D3DXCreateBox(d3ddev, 1.0f, 1.0f, 1.0f, &mesh, NULL);
```

### 2. 创建球体

使用 D3DXCreateSphere 函数可以在运行时动态创建一个球体。图 13-2 显示了该程序的这一版本的输出。

```
D3DXCreateSphere(d3ddev, 1.0f, 20, 20, &mesh, NULL);
```

### 3. 创建圆柱体

圆柱体要比前面两种后援对象复杂一些，它的创建可使用 D3DXCreateCylinder 函数，图 13-3 给出了其输出。

```
D3DXCreateCylinder(d3ddev, 1.0f, 1.0f, 2.0f, 20, 20, &mesh, NULL);
```

### 4. 创建面包圈

面包圈是一种圆环或者轮管状的对象，可使用 D3DXCreateTorus 函数来创建它。这是笔者最喜欢的后援网格，因为它很好地演示了光照（在完全点亮的环境中）。图 13-4 给出了其输出。

```
D3DXCreateTorus(d3ddev, 0.5f, 1.0f, 20, 20, &mesh, NULL);
```

### 5. 创建茶壶

最后，我们可以动态创建一个真实世界的对象：茶壶。多年来，这个网格是成百上千本图书、文件和网站介绍的主题，它有点明星范儿。图 13-5 给出了其输出。

```
ID3DXCreateTeapot(d3ddev, &mesh, NULL);
```

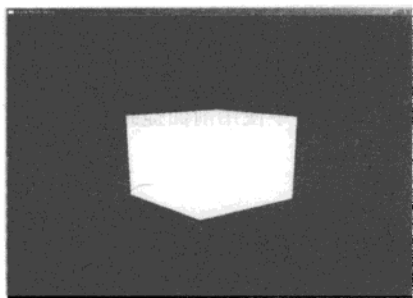


图 13-1 渲染一个以后援网格生成的立方体

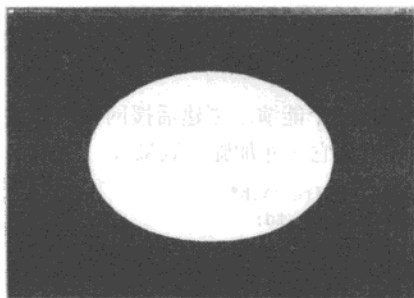


图 13-2 渲染一个以后援网格生成的球体

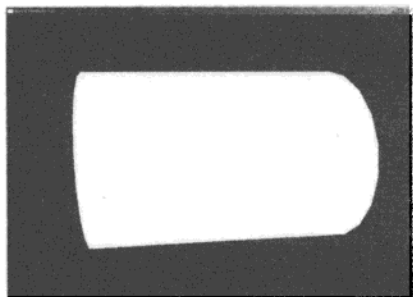


图 13-3 渲染一个以后援网格生成的圆柱体

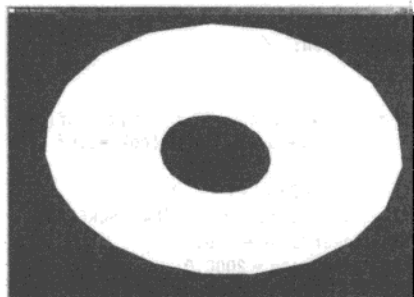


图 13-4 渲染一个以后援网格生成的面包圈

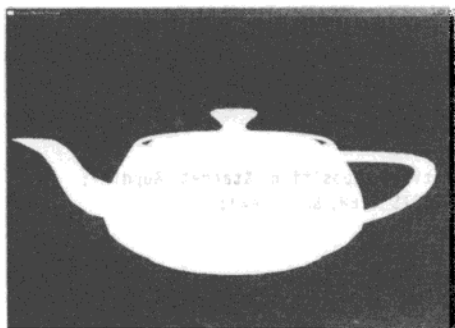


图 13-5 渲染一个以后援网格生成的茶壶

### 13.1.2 渲染后援网格

虽然每种后援网格都由各自的需要不同参数集的函数来创建，但所有的后援网格函数都将顶点数据填入同样的 ID3DXMesh 对象中。这就意味着这些网格在创建之后都会被以相同的方式来

处理。如果不关心材质或纹理，那么网格的渲染可以使用 `ID3DXMesh::DrawSubset` 函数来实现：

```
mesh->DrawSubset(0);
```

### 13.1.3 Stock\_Mesh 程序

下面创建一个能演示创建后援网格的方法的示例，在程序中使用前面提到的其中一个函数创建网格然后渲染它（外加旋转效果）。

```
#include "MyDirectX.h"
using namespace std;

const string APPTITLE = "Stock Mesh Demo";
const int SCREENW = 1024;
const int SCREENH = 768;

DWORD screentimer = 0;

LPD3DXMESH mesh;

void SetCamera(float posx, float posy, float posz,
               float lookx = 0.0f, float looky = 0.0f, float lookz = 0.0f)
{
    float fov = D3DX_PI / 4.0;
    float aspectRatio = SCREENW / SCREENH;
    float nearRange = 1.0;
    float farRange = 2000.0;
    D3DXVECTOR3 updir = D3DXVECTOR3(0.0f, 1.0f, 0.0f);
    D3DXVECTOR3 position = D3DXVECTOR3(posx, posy, posz);
    D3DXVECTOR3 target = D3DXVECTOR3(lookx, looky, lookz);

    //set the perspective
    D3DXMATRIX matProj;
    D3DXMatrixPerspectiveFovLH(&matProj, fov, aspectRatio, nearRange, farRange);
    d3ddev->SetTransform(D3DTS_PROJECTION, &matProj);

    //set up the camera view matrix
    D3DXMATRIX matView;
    D3DXMatrixLookAtLH(&matView, &position, &target, &updir);
    d3ddev->SetTransform(D3DTS_VIEW, &matView);
}

bool Game_Init(HWND window)
{
    //initialize Direct3D
    if (!Direct3D_Init(window, SCREENW, SCREENH, false))
    {
        MessageBox(window, "Error initializing Direct3D", APPTITLE.c_str(), 0);
        return false;
    }

    //initialize DirectInput
    if (!DirectInput_Init(window))
    {
```

```

    MessageBox(window, "Error initializing DirectInput",
        APPTITLE.c_str(), 0);
    return false;
}

//initialize DirectSound
if (!DirectSound_Init(window))
{
    MessageBox(window, "Error initializing DirectSound",
        APPTITLE.c_str(), 0);
    return false;
}

//set the camera position
SetCamera( 0.0f, 1.0f, -4.0f );

//use ambient lighting and z-buffering
d3ddev->SetRenderState(D3DRS_ZENABLE, true);
d3ddev->SetRenderState(D3DRS_LIGHTING, false);

/**
 * create a stock 3D mesh: uncomment only one line at a time
 * and re-run the program to see each shape separately
 */
//D3DXCreateBox(d3ddev, 1.0f, 1.0f, 1.0f, &mesh, NULL);
//D3DXCreateSphere(d3ddev, 1.0f, 20, 20, &mesh, NULL);
//D3DXCreateCylinder(d3ddev, 1.0f, 1.0f, 2.0f, 20, 20, &mesh, NULL);
D3DXCreateTorus(d3ddev, 0.5f, 1.0f, 20, 20, &mesh, NULL);
//D3DXCreateTeapot(d3ddev, &mesh, NULL);

return true;
}

void Game_Run(HWND window)
{
    if (!d3ddev) return;
    DirectInput_Update();
    d3ddev->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
        D3DCOLOR_XRGB(0, 0, 100), 1.0f, 0);

    // slow rendering to approximately 60 fps
    if (timeGetTime() > screentimer + 14)
    {
        screentimer = GetTickCount();

        //start rendering
        if (d3ddev->BeginScene())
        {
            //rotate the view
            D3DXMATRIX matWorld;
            D3DXMatrixRotationY(&matWorld, timeGetTime()/1000.0f);
            d3ddev->SetTransform(D3DTS_WORLD, &matWorld);
        }
    }
}

```





```

        //draw the mesh
        mesh->DrawSubset(0);

        //stop rendering
        d3ddev->EndScene();
        d3ddev->Present(NULL, NULL, NULL, NULL);
    }

    //exit with escape key or controller Back button
    if (KEY_DOWN(VK_ESCAPE)) gameover = true;
    if (controllers[0].wButtons & XINPUT_GAMEPAD_BACK) gameover = true;
}

void Game_End()
{
    //free memory and shut down
    mesh->Release();

    DirectSound_Shutdown();
    DirectInput_Shutdown();
    Direct3D_Shutdown();
}

```

## 13.2 装载并渲染模型文件

在示例或游戏中使用后援网格非常方便。例如，在滚动发射器中使用小球体作为子弹！而我们接下来需要学习如何从网格文件中读入 3D 模型然后渲染它。准备好了吗？我们开始吧。

### 13.2.1 装载 .X 文件

Direct3D 提供一个从已装载的 .X 文件中创建网格的函数，于是，要将任何一个模型文件读入我们的游戏中，将非常简单。我们慢慢来，详细探究每一个步骤，然后在本章的末尾给出一组可重用的函数。

#### 1. 定义新的 MODEL 结构

首先，我们需要一个新的结构来处理要装载的模型文件：

```

struct MODEL
{
    LPD3DXMESH mesh;
    D3DMATERIAL9* materials;
    LPDIRECT3DTEXTURE9* textures;
    DWORD material_count;
};

```

有些程序员和建模师称它们为“网格”文件，但笔者使用更具描述性的“3D 模型”的说法，因为这样更易于初学者理解。MODEL 结构包含需要装载及渲染一个模型文件所需的主对象。首先，我们有网格数据（由定点组成）。然后，有一个 D3DMATERIAL9 指针变量将会装载在模型

文件中定义的材质数组。在处理过精灵之后，读者应该已经熟悉了 LPDIRECT3DTEXTURE9，所以这里没有惊喜，只是模型可能使用多个纹理。这些纹理并不存储于模型文件自身中，而是在分开的位图文件中，在模型文件中存储的仅仅是纹理的文件名。

最后，有一个成员变量保存模型中材质的数量，在渲染时要用到它。在模型中可以有许多材质，但不是每一个材质都需要有纹理。不过，纹理必须定义于材质之内。于是，我们有一个 material\_count 变量，但这里却无需保存纹理的数量。

## 2. 装载网格

装载模型文件的关键在于 D3DXLoadMeshFromX 函数：

```
HRESULT WINAPI D3DXLoadMeshFromX(
    LPCTSTR pFilename,
    DWORD Options,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXBUFFER *ppAdjacency,
    LPD3DXBUFFER *ppMaterials,
    LPD3DXBUFFER *ppEffectInstances,
    DWORD *pNumMaterials,
    LPD3DXMESH *ppMesh
);
```

这个函数的参数不是使用默认值（以某种形式）就是 NULL，其关键参数有文件名、Direct3D 设备、材质缓冲区、材质数量和网格对象。首先需要有个材质缓冲区用于装载材质：

```
LPD3DXBUFFER matbuffer;
```

我们也假设指向 MODEL 结构的指针已经创建好了：

```
MODEL *model = (MODEL*)malloc(sizeof(MODEL));
```

这个结构位于内存中，由 LoadModel 函数返回（稍后讲解这个函数）。然后可以读入模型文件并同时装载材质和网格。以下是调用这个函数的示例代码：

```
result = D3DXLoadMeshFromX(
    filename,           //filename
    D3DXMESH_SYSTEMMEM, //mesh options
    d3ddev,            //Direct3D device
    NULL,              //adjacency buffer
    &matbuffer,         //material buffer
    NULL,              //special effects
    &model->material_count, //number of materials
    &model->mesh);      //resulting mesh
```

## 3. 装载材质和纹理

材质存储于材质缓冲区中，不过，在渲染模型之前需要将它们转换成 Direct3D 材质和纹理。我们熟悉纹理对象，但材质对象——LPD3DXMATERIAL 则是新的。

以下是从材质缓冲区中将材质和纹理复制到各个材质和纹理数组中的方法。首先，创建数组：

```
D3DMATERIAL* d3dxMaterials = (LPD3DMATERIAL)matbuffer->GetBufferPointer();
model->materials = new D3DMATERIAL9[model->material_count];
model->textures = new LPDIRECT3DTEXTURE9[model->material_count];
```

下一步是迭代材质并将它们从材质缓冲区中取出。对于每个材质，都会设置环境颜色，都会将纹理装载到纹理对象中。由于这些是动态分配的数组，所以一个模型仅受限于可用内存及渲染它的显卡的能力。我们可以有一个具备上百万张面的模型，每张面都有不同的材质。

```
//create the materials and textures
for(DWORD i=0; i<model->material_count; i++)
{
    //grab the material
    model->materials[i] = d3dxMaterials[i].MatD3D;

    //set ambient color for material
    model->materials[i].Ambient = model->materials[i].Diffuse;

    model->textures[i] = NULL;
    if (d3dxMaterials[i].pTextureFilename != NULL)
    {
        string filename = d3dxMaterials[i].pTextureFilename;
        if( FindFile(&filename) )
        {
            result = D3DXCreateTextureFromFile(
                d3ddev, filename.c_str(), &model->textures[i]);

            if (result != D3D_OK) {
                MessageBox(0, "Could not find texture", APPTITLE.c_str(), 0);
                return false;
            }
        }
    }
}
```

你是否注意到在上面的代码清单中有一个不认识的函数调用 FindFile？如果没有，那么你真该集中注意力了！这是个助手函数，它对于在 Direct3D 中装载纹理实在是很重要。常见的是，网格文件嵌入了带有完整路径名且硬编码了的纹理文件名。在装载网格文件并试着分析纹理文件名时，我们将会得到硬编码了的、表示建模师计算机系统上的路径名，这对我们的游戏项目毫无意义。

（这是一个因为 Maya 和 3ds max 存储纹理文件名的方法而带来的非常常见的问题，除非建模师手工修改文件名。）所以，我们必须向这个问题妥协。例如，图 13-6 显示了装载到 DirectX Viewer 中的 Fokker 飞机模型文件。注意，它没有纹理！这是因为纹理文件使用了硬编码的路径。

由 Fokker.x 文件引用的纹理文件是 Fokker.bmp，图 13-7 显示了该纹理。注意，大多数 .X 文件是二进制文件，所以无法打开它们并且编辑纹理路径名！虽然 Direct3D 的确支持文本版的 .X 文件格式，但却很少用到。

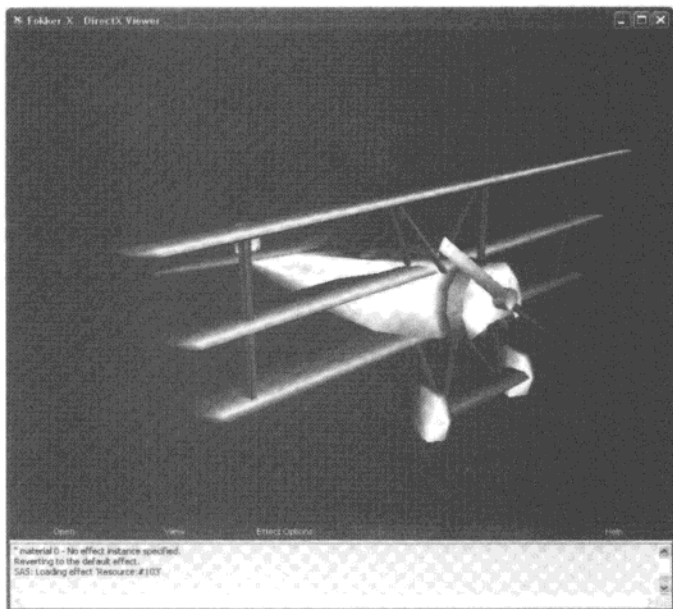


图 13-6 在 DirectX Viewer 程序中查看 Fokker 飞机模型

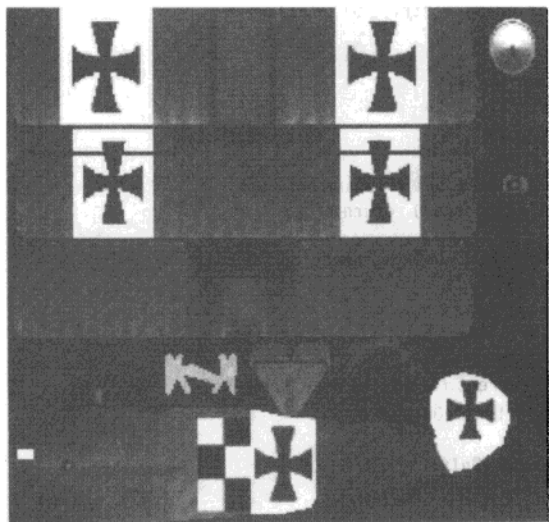


图 13-7 Fokker 飞机模型的纹理

在网格中遇到一个文件名时，为了定位纹理文件，需要三个函数一起工作。首先，我们有 FindFile 函数，它嵌入于 LoadMesh 函数中。FindFile 函数也需要两个助手函数：DoesFileExist 和 SplitPath，它们的功能如它们的名称表示的，分别表示文件是否存在和路径分割。

```
void SplitPath(const string& inputPath, string* pathOnly, string* filenameOnly)
{
    string fullPath( inputPath );
    replace( fullPath.begin(), fullPath.end(), '\\', '/' );
    string::size_type lastSlashPos = fullPath.find_last_of('/');

    // check for there being no path element in the input
    if (lastSlashPos == string::npos)
    {
        *pathOnly="";
        *filenameOnly = fullPath;
    }
    else {
        if (pathOnly) {
            *pathOnly = fullPath.substr(0, lastSlashPos);
        }
        if (filenameOnly)
        {
            *filenameOnly = fullPath.substr(
                lastSlashPos + 1,
                fullPath.size() - lastSlashPos - 1 );
        }
    }
}

bool DoesFileExist(const string &filename)
{
    return (_access(filename.c_str(), 0) != -1);
}

bool FindFile(string *filename)
{
    if (!filename) return false;

    //look for file using original filename and path
    if (DoesFileExist(*filename)) return true;

    //since the file was not found, try removing the path
    string pathOnly;
    string filenameOnly;
    SplitPath( *filename, &pathOnly, &filenameOnly);

    //is file found in current folder, without the path?
    if (DoesFileExist(filenameOnly))
    {
        *filename=filenameOnly;
        return true;
    }

    //not found
    return false;
}
```



### 13.2.2 渲染完整的模型

在装载了模型之后，绘制它就是小菜一碟了。但是装载模型的代码如果不一行一行读上好几遍的话，不是那么容易理解的。而渲染的代码则容易得多，应该很好理解，因为我们已经用过 DrawPrimitive 函数了。还记得第12章的 Textured\_Cube 程序吗？那就是一个简单的渲染3D模型的示例，我们现在要做的也是一样。

首先设置材质和纹理，然后调用 DrawPrimitive 函数来显示多边形（面）。最大的不同在于我们现在必须使用 material\_count 的值对模型进行迭代并使用 DrawSubset 函数渲染每一个面。这段代码足够智能，它可以跳过不存在的材质。以下是它的工作原理：

```
//any materials in this mesh?
if (model->material_count == 0)
{
    model->mesh->DrawSubset(0);
}
else {
    //draw each mesh subset
    for( DWORD i=0; i < model->material_count; i++)
    {
        // Set the material and texture for this subset
        d3ddev->SetMaterial( &model->materials[i] );

        if (model->textures[i])
        {
            if (model->textures[i]->GetType() == D3DRTYPE_TEXTURE)
            {
                D3DSURFACE_DESC desc;
                model->textures[i]->GetLevelDesc(0, &desc);
                if (desc.Width > 0) {
                    d3ddev->SetTexture( 0, model->textures[i] );
                }
            }
        }

        // Draw the mesh subset
        model->mesh->DrawSubset( i );
    }
}
```

### 13.2.3 从内存中删除一个模型

在使用完一个 MODEL 对象之后，需要将其资源释放，否则会给程序带来内存泄漏。游戏中所用的资源通常在游戏结束时释放。

```
//remove materials from memory
if( model->materials != NULL )
    delete[] model->materials;

//remove textures from memory
if (model->textures != NULL)
```

```
(  
    for( DWORD i = 0; i < model->material_count; i++)  
    {  
        if (model->textures[i] != NULL)  
            model->textures[i]->Release();  
    }  
    delete[] model->textures;  
)  
  
//remove mesh from memory  
if (model->mesh != NULL)  
    model->mesh->Release();  
  
//remove model struct from memory  
if (model != NULL) free(model);
```

### 13.2.4 Render\_Mesh 程序

笔者编写了一个完整的装载网格文件（以 .X 为扩展名）并且以完全着色的方式在屏幕上渲染的程序，下面将讲解这个程序。图 13-8 显示了运行中的 Render\_Mesh 程序。是不是很酷啊？

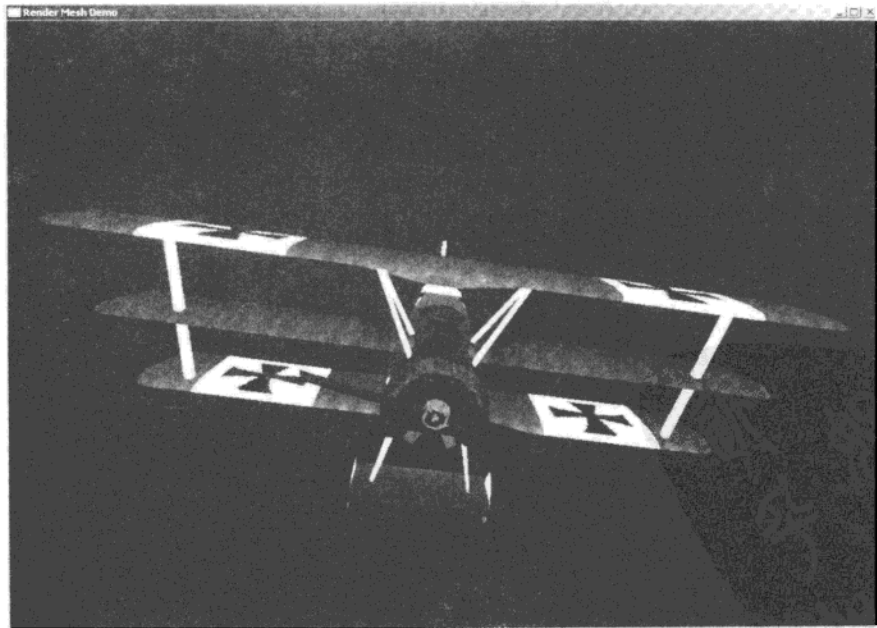


图 13-8 Render\_Mesh 程序从 .X 文件中装载并渲染网格

#### 1. 修改 MyDirectX.h

由于又有了一些可重用的 3D 网格装载和渲染代码，所以我们现在要在 MyDirectX 文件中添

加一些新功能。这些代码可以添加到文件的底部。这些 DirectX 助手文件增长得有些大，包含了来自不同组件的代码，不过这样很方便。注意，MODEL 结构从我们在本章所看的第一个示例开始到现在已经变化了很多，它还包含一个构造函数用于初始化其属性变量（注意，CD 光盘上的 DirectX\_Project 模板中已经添加了 MODEL 结构和相关函数原型）。

```
//define the MODEL struct
struct MODEL
{
    LPD3DXMESH mesh;
    D3DMATERIAL9* materials;
    LPDIRECT3DTEXTURE9* textures;
    DWORD material_count;
    D3DXVECTOR3 translate;
    D3DXVECTOR3 rotate;
    D3DXVECTOR3 scale;

    MODEL()
    {
        material_count = 0;
        mesh = NULL;
        materials = NULL;
        textures = NULL;
        translate = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
        rotate = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
        scale = D3DXVECTOR3(1.0f, 1.0f, 1.0f);
    }
};

//3D mesh function prototypes
void DrawModel(MODEL *model);
void DeleteModel(MODEL *model);
MODEL *LoadModel(string filename);
bool FindFile(string *filename);
bool DoesFileExist(const string &filename);
void SplitPath(const string& inputPath, string* pathOnly, string* filenameOnly);
void SetCamera(float posX, float posY, float posz,
    float lookx = 0.0f, float looky = 0.0f, float lookz = 0.0f);
```

## 2. 修改 MyDirectX.cpp

我们现在必须对 MyDirectX.cpp 文件进行如下增加。这些主要都是从已有的、已经向读者介绍过的代码构建的可重用函数。将这些代码添加到 MyDirectX.cpp 或者打开已建立的项目查看最后的结果（注意，CD 光盘上的 DirectX\_Project 模板中已经添加了这些函数）。

```
void SetCamera(float posX, float posY, float posz,
    float lookx, float looky, float lookz)
{
    float fov = D3DX_PI / 4.0;
    float aspectRatio = SCREENW / SCREENH;
    float nearRange = 1.0;
    float farRange = 2000.0;
```



```

D3DXVECTOR3 updir = D3DXVECTOR3(0.0f, 1.0f, 0.0f);
D3DXVECTOR3 position = D3DXVECTOR3(posx, posy, posz);
D3DXVECTOR3 target = D3DXVECTOR3(lookx, looky, lookz);

//set the perspective
D3DXMATRIX matProj;
D3DXMatrixPerspectiveFovLH(&matProj, fov, aspectRatio, nearRange,
farRange);
d3ddev->SetTransform(D3DTS_PROJECTION, &matProj);

//set up the camera view matrix
D3DXMATRIX matView;
D3DXMatrixLookAtLH(&matView, &position, &target, &updir);
d3ddev->SetTransform(D3DTS_VIEW, &matView);
}

void SplitPath(const string& inputPath, string* pathOnly, string* filenameOnly)
{
    string fullPath( inputPath );
    replace( fullPath.begin(), fullPath.end(), '\\', '/' );
    string::size_type lastSlashPos = fullPath.find_last_of('/');

    // check for there being no path element in the input
    if (lastSlashPos == string::npos)
    {
        *pathOnly="";
        *filenameOnly = fullPath;
    }
    else {
        if (pathOnly) {
            *pathOnly = fullPath.substr(0, lastSlashPos);
        }
        if (filenameOnly)
        {
            *filenameOnly = fullPath.substr(
                lastSlashPos + 1,
                fullPath.size() - lastSlashPos - 1 );
        }
    }
}

bool DoesFileExist(const string &filename)
{
    return (_access(filename.c_str(), 0) != -1);
}

bool FindFile(string *filename)
{
    if (!filename) return false;

    //look for file using original filename and path
    if (DoesFileExist(*filename)) return true;
}

```



```

//since the file was not found, try removing the path
string pathOnly;
string filenameOnly;
SplitPath(*filename,&pathOnly,&filenameOnly);

//is file found in current folder, without the path?
if (DoesFileExist(filenameOnly))
{
    *filename=filenameOnly;
    return true;
}

//not found
return false;
}

MODEL *LoadModel(string filename)
{
    MODEL *model = (MODEL*)malloc(sizeof(MODEL));
    LPD3DXBUFFER matbuffer;
    HRESULT result;

    //load mesh from the specified file
    result = D3DXLoadMeshFromX(
        filename.c_str(),           //filename
        D3DXMESH_SYSTEMMEM,        //mesh options
        d3ddev,                    //Direct3D device
        NULL,                      //adjacency buffer
        &matbuffer,                 //material buffer
        NULL,                      //special effects
        &model->material_count, //number of materials
        &model->mesh);             //resulting mesh

    if (result != D3D_OK)
    {
        MessageBox(0, "Error loading model file", APPTITLE.c_str(), 0);
        return NULL;
    }

    //extract material properties and texture names from material buffer
    LPD3DXMATERIAL d3dxMaterials = (LPD3DXMATERIAL)matbuffer->
    GetBufferPointer();
    model->materials = new D3DMATERIAL9[model->material_count];
    model->textures = new LPDIRECT3DTEXTURE9[model->material_count];

    //create the materials and textures
    for(DWORD i=0; i<model->material_count; i++)
    {
        //grab the material
        model->materials[i] = d3dxMaterials[i].MatD3D;

        //set ambient color for material
        model->materials[i].Ambient = model->materials[i].Diffuse;
    }
}

```

```
model->textures[i] = NULL;
if (d3dxMaterials[i].pTextureFilename != NULL)
{
    string filename = d3dxMaterials[i].pTextureFilename;
    if( FindFile(&filename) )
    {
        result = D3DXCreateTextureFromFile(
            d3ddev, filename.c_str(), &model->textures[i]);
        if (result != D3D_OK)
        {
            MessageBox(0, "Could not find texture",
                APPTITLE.c_str(), 0);
            return false;
        }
    }
}
}

//done using material buffer
matbuffer->Release();

return model;
}

void DeleteModel(MODEL *model)
{
    //remove materials from memory
    if( model->materials != NULL )
        delete[] model->materials;

    //remove textures from memory
    if (model->textures != NULL)
    {
        for( DWORD i = 0; i < model->material_count; i++)
        {
            if (model->textures[i] != NULL)
                model->textures[i]->Release();
        }
        delete[] model->textures;
    }

    //remove mesh from memory
    if (model->mesh != NULL)
        model->mesh->Release();

    //remove model struct from memory
    if (model != NULL)
        free(model);
}

void DrawModel(MODEL *model)
{
    //any materials in this mesh?
    if (model->material_count == 0)
```



```

{
    model->mesh->DrawSubset(0);
}
else {
    //draw each mesh subset
    for( DWORD i=0; i < model->material_count; i++ )
    {
        // Set the material and texture for this subset
        d3ddev->SetMaterial( &model->materials[i] );

        if (model->textures[i])
        {
            if (model->textures[i]->GetType() == D3DRTYPE_TEXTURE)
            {
                D3DSURFACE_DESC desc;
                model->textures[i]->GetLevelDesc(0, &desc);
                if (desc.Width > 0) {
                    d3ddev->SetTexture( 0, model->textures[i] );
                }
            }
        }

        // Draw the mesh subset
        model->mesh->DrawSubset( i );
    }
}
}

```

在更新了 MyDirectX 文件，添加了可重用的网格代码之后，我们可以解决这个示例程序的主源代码的问题了。

```

#include "MyDirectX.h"
using namespace std;

const string APPTITLE = "Render Mesh Demo";
const int SCREENW = 1024;
const int SCREENH = 768;

DWORD screentimer = timeGetTime();

MODEL *mesh=NULL;

bool Game_Init(HWND window)
{
    srand( (int)time(NULL) );

    //initialize Direct3D
    if (!IDirect3D_Init(window, SCREENW, SCREENH, false))
    {
        MessageBox(window, "Error initializing Direct3D", APPTITLE.c_str(), 0);
        return false;
    }
}

```



```
//initialize DirectInput
if (!DirectInput_Init(window))
{
    MessageBox(window, "Error initializing DirectInput",
        APPTITLE.c_str(), 0);
    return false;
}

//initialize DirectSound
if (!DirectSound_Init(window))
{
    MessageBox(window, "Error initializing DirectSound",
        APPTITLE.c_str(), 0);
    return false;
}

//set the camera position
SetCamera( 0.0f, -800.0f, -200.0f );

//use ambient lighting and z-buffering
d3ddev->SetRenderState(D3DRS_ZENABLE, true);
d3ddev->SetRenderState(D3DRS_LIGHTING, false);

//load the mesh file
mesh = LoadModel("Fokker.X");
if (mesh == NULL)
{
    MessageBox(window, "Error loading mesh", APPTITLE.c_str(), MB_OK);
    return 0;
}

return true;
}

void Game_Run(HWND window)
{
    if (!d3ddev) return;
    DirectInput_Update();
    d3ddev->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
        D3DCOLOR_XRGB(0,0,100), 1.0f, 0);

    /*
    * slow rendering to approximately 60 fps
    */
    if (timeGetTime() > screentimer + 14)
    {
        screentimer = GetTickCount();

        //start rendering
        if (d3ddev->BeginScene())
        {
            //rotate the view
            D3DXMATRIX matWorld;
```

```

D3DXMatrixRotationY(&matWorld, timeGetTime()/1000.0f);
d3ddev->SetTransform(D3DTS_WORLD, &matWorld);

//draw the model
DrawModel(mesh);

//stop rendering
d3ddev->EndScene();
d3ddev->Present(NULL, NULL, NULL, NULL);
}

}

//exit with escape key or controller Back button
if (KEY_DOWN(VK_ESCAPE)) gameover = true;
if (controllers[0].wButtons & XINPUT_GAMEPAD_BACK) gameover = true;
}

void Game_End()
{
    //free memory and shut down
    DeleteModel(mesh);

    DirectSound_Shutdown();
    DirectInput_Shutdown();
    Direct3D_Shutdown();
}

```

我们现在有了将网格文件装载到我们自己的游戏中的能力了！于是，头脑中的很多想法都有可能实现了。天空是唯一的限制！任何我们能想得到的 3D 游戏，我们都有能力来实现了。当然，在这个过程中有许多细节需要补齐，但这已经是非常好的开始了。如同往常一样，这里已经为你提供了一个马上可以使用的项目供你修改（见 CD-ROM 上的 \sources\chapter13 中的 DirectX\_Project）。

### 13.3 你所学到的

本章提供了在 Direct3D 中将模型文件装载到内存并渲染所需的信息！有以下几个要点：

- 如何在运行时创建及渲染后援对象。
- 如何从 X 文件中装载并渲染网格。

### 13.4 复习测验

以下复习测验题有助于考查你是否已经掌握了本章中的所有内容。

- 1) 表示网格的 Direct3D 对象的名称是什么？
- 2) 随着程序对材质进行迭代而一个一个渲染网格中的每个面的 Direct3D 函数是哪个？
- 3) 我们可用于将 X 文件装载到 Direct3D 网格中的函数的名称是什么？
- 4) 用于绘制模型中各个多边形的函数是哪个？

- 5) 用于表示内存中的纹理的 Direct3D 数据类型是什么?
- 6) 用于储存矩阵的 Direct3D 数据类型的名称是什么?
- 7) 哪个 Direct3D 函数在 Y 轴上旋转网格?
- 8) 哪个标准矩阵通常表示场景中当前被转换及渲染的对象?
- 9) 用于指定诸如长宽比这样的属性的标准矩阵是哪个?
- 10) 哪个标准矩阵表示照相机视图?

## 13.5 自己动手

以下习题将帮助读者学习更多与本章有关的知识。

习题 1 修改 Stock\_Mesh 程序, 让它在屏幕上同时绘制两个后援网格。

习题 2 Render\_Mesh 程序演示了装载 X 文件及在屏幕上渲染它的方法。修改这个程序, 让它使用键盘或鼠标来旋转模型, 而不仅仅让用户看着它自己旋转。



## 第三部分 >>>

### 游戏项目

---

这一部分只包含一章，它给出了一个近乎完整的游戏以供读者进行学习、修改、调整及实验之用，学习在屏幕上同时有大量不同的且移且动而且交互的对象的大规模游戏的构造方法。





## 第 14 章 Anti-Virus（反病毒）游戏

本章专注于一个能有效地演示在前面 13 章中所学的概念的游戏项目！这个游戏是个原型，或者说还是一个在进行过程中的游戏，我们特意将它停留在一个简单的状态以便读者可以不被高级游戏功能的复杂性所打搅，从而学习其精髓，否则这个游戏项目按我的经验可能有 10 倍于现在的大小。这个游戏原型只有大约 1 100 行留白良好并且良好注释的代码。鼓励读者学习本章给出并且讲解的代码，然后打开 CD 光盘上的项目，以你自己的想法来改进这一游戏！

本章将学到：

- 创建游戏项目。
- 编写源代码。

### 14.1 Anti-Virus 游戏

这里特意将本章的主角——Anti-Virus 游戏的细节和函数分散开来，以便读者能增强它！本游戏有一个滚动背景、一个每帧转换并绘制上百个精灵的高速游戏循环、对键盘和 Xbox 360 控制器的支持及一些非常有趣的音响效果！

**建议** Anti-Virus 游戏中的所有音响效果都是使用 Audacity 这个声音编辑软件从正弦波、方波和锯齿波生成的。在玩这个游戏时别忘了把扬声器打开！

Anti-Virus 游戏有一个基本的故事和情节，但没有正式的游戏设计文档，所以，这里简要解释一下故事。如果要将这个伪游戏转变为正式的游戏项目，那么需要来一次 30 秒的“电梯推介”：

美国宇航局负责处理火星探测器的超级计算机被发送自探测器（通过它们传回给地球的无线电信号）的外星计算机病毒所入侵。这种外星病毒是一种人类从来没有遇到过的病毒类型，它们的行为似乎更像生命体而不是计算机程序。

在被下载之后，外星病毒能够自我翻译成我们的计算机系统能理解的形式。它们现在正在销毁存储在超级计算机中的火星数据！宇航局的工程师们担心，这种外星计算机病毒可能扩散到其他系统中，于是隔离了受感染的超级计算机。所有已知的反病毒软件和网络安全策略都不管用！

但现在情况不同了！我们有新一代的纳米机器人技术，用它来构造一种微型的遥控机器人。你的使命很简单：进入超级计算机的核心内存并消灭外星威胁！

你的纳米机器人的代号是“R.A.I.N.e.R.”：遥控人工智能纳米机器人（Remote-controlled Artificially Intelligent Nano-Robot）。RAINeR 是一个完全独立的、自立的机器，但如果需要持续长时间运行需要有能量，尤其是在安装了附加的病毒消灭装备之后。核心计算机内存中填满了老程序和数据的片段，这些可以用于给纳米机器人重新充电。

你的使命很简单：通过备用通信线路进入超级计算机的内存，穿过防火墙和内存平台进入计算机的核心。一旦到达，你将销毁第一个外星病毒——“母毒”，然后获得其身份代码。只有到这个时候我们才能构建一个防御网来抵抗病毒的传播。

祝你好运！

### 14.1.1 游戏玩法

Anti-Virus 游戏已经有许多有趣的游戏功能，只需连接在一起就可带来协同体验。例如，游戏有对武器升级的支持，但当前还没有提供游戏者可以选择的任何升级能力。要想更改纳米机器人的火力升级水平，可按 F1 到 F5 键，F1 是默认的武器而 F5 则是完全升级的版本。

#### 1. 火力 1

图 14-1 显示了第一种武器，这是默认武器。

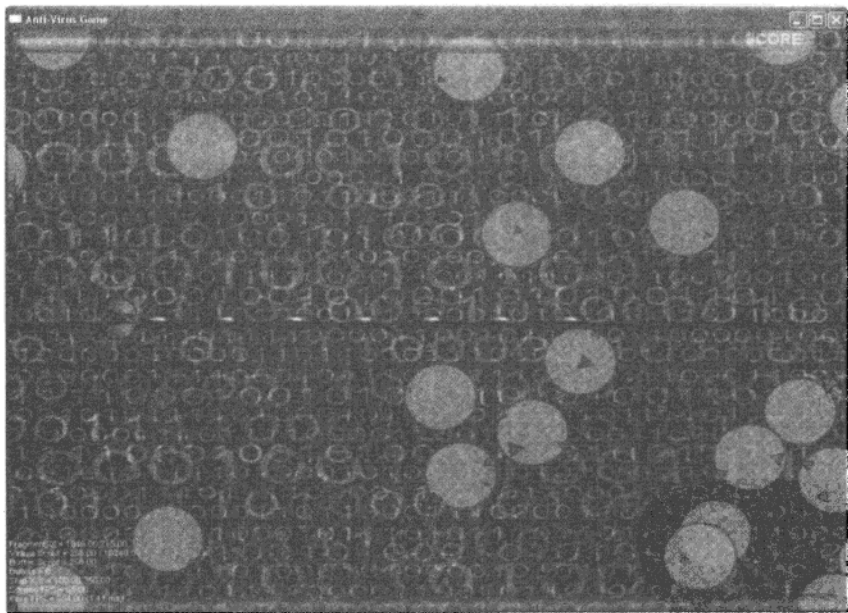


图 14-1 以正常火力一次发一个单发子弹

即使只想打一个单发子弹，我们仍旧需要实现一个子弹数组并且要留心进行计时，以免子弹就如火龙一样喷出。打子弹的代码可在 `player_shoot()` 函数中找到，这里有一个 `switch` 语句用于以纳米机器人的火力级别为基础来触发相应代码。

```
case 1:
{
    //create a bullet
    int b1 = find_bullet();
    if (b1 == -1) return;
```

```
bullets[b1].alive = true;
bullets[b1].rotation = 0.0;
bullets[b1].velx = 12.0f;
bullets[b1].vely = 0.0f;
bullets[b1].x = player.x + player.width/2;
bullets[b1].y = player.y + player.height/2
    - bullets[b1].height/2;
}
break;
```

子弹以相对于游戏者的位置来定位，并且有一点点调整，以便子弹正好出现在纳米机器人中央的前方。其他四种火力的变化也是基于这段代码。图形用户界面（GUI）有点松散，但已经有模有样了，在顶端有能量条，计算机的健康状态条则位于屏幕的底部。随着游戏的进行，外星病毒将会对计算机进行破坏，用户不仅要击退外星威胁而且要维修病毒给计算机系统造成的破坏（可以通过短的迷你游戏或者必须要收集的对象来进行，这里有无可能）。

## 2. 火力 2

第二火力级别如图 14-2 所示，它的特点是有两颗子弹从纳米机器人朝向敌方的病毒程序（以让人想起生物细胞的半透明圆圈来表示）发出。可使用 F2 调试键来装备这个级别。

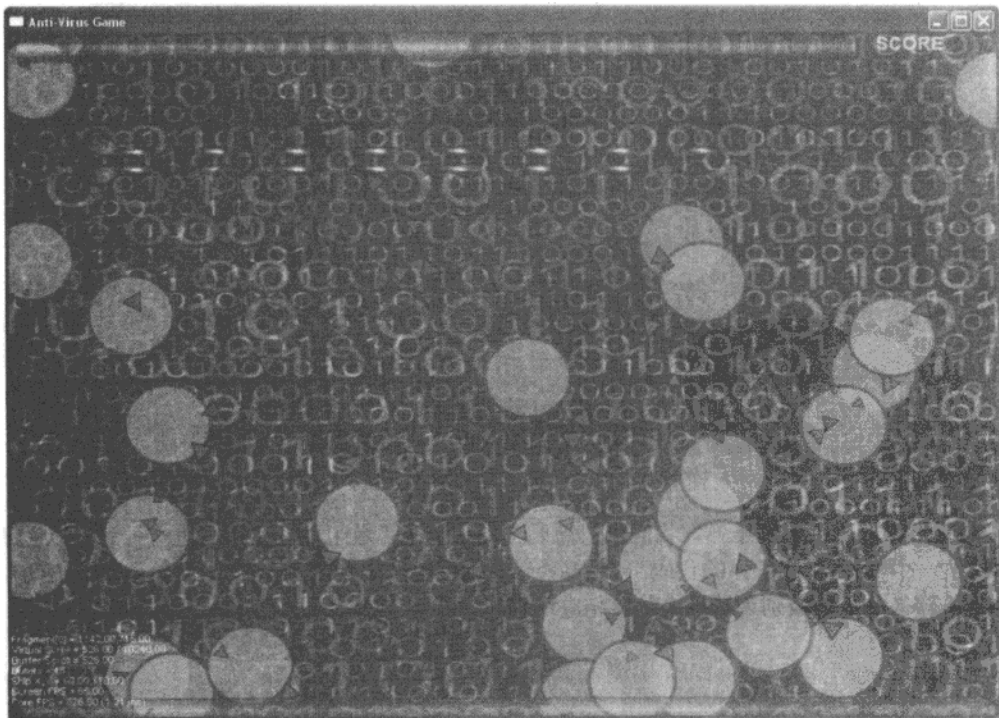


图 14-2 使用火力级别 2 一次发出两发子弹

```

case 2:
{
    //create bullet 1
    int b1 = find_bullet();
    if (b1 == -1) return;
    bullets[b1].alive = true;
    bullets[b1].rotation = 0.0;
    bullets[b1].velx = 12.0f;
    bullets[b1].vely = 0.0f;
    bullets[b1].x = player.x + player.width/2;
    bullets[b1].y = player.y + player.height/2
        - bullets[b1].height/2;
    bullets[b1].y -= 10;

    //create bullet 2
    int b2 = find_bullet();
    if (b2 == -1) return;
    bullets[b2].alive = true;
    bullets[b2].rotation = 0.0;
    bullets[b2].velx = 12.0f;
    bullets[b2].vely = 0.0f;
    bullets[b2].x = player.x + player.width/2;
    bullets[b2].y = player.y + player.height/2
        - bullets[b2].height/2;
    bullets[b2].y += 10;
}
break;

```

### 3. 火力 3

第三个火力级别见图 14-3, 它的特点是有三颗子弹从纳米机器人朝向敌方的病毒程序发出。第二火力级别有两颗子弹以离纳米机器人同样的距离出现, 而第三级则是有一颗中心子弹和上一下另外两颗子弹。学习对每颗子弹进行定位的代码可理解如何通过修改这些火力级别来实现我们自己自定义的子弹配置。

```

case 3:
{
    //create bullet 1
    int b1 = find_bullet();
    if (b1 == -1) return;
    bullets[b1].alive = true;
    bullets[b1].rotation = 0.0;
    bullets[b1].velx = 12.0f;
    bullets[b1].vely = 0.0f;
    bullets[b1].x = player.x + player.width/2;
    bullets[b1].y = player.y + player.height/2
        - bullets[b1].height/2;

    //create bullet 2
    int b2 = find_bullet();
    if (b2 == -1) return;
    bullets[b2].alive = true;
    bullets[b2].rotation = 0.0;

```



```

bullets[b2].velx = 12.0f;
bullets[b2].vely = 0.0f;
bullets[b2].x = player.x + player.width/2;
bullets[b2].y = player.y + player.height/2
    - bullets[b2].height/2;
bullets[b2].y -= 16;

//create bullet 3
int b3 = find_bullet();
if (b3 == -1) return;
bullets[b3].alive = true;
bullets[b3].rotation = 0.0;
bullets[b3].velx = 12.0f;
bullets[b3].vely = 0.0f;
bullets[b3].x = player.x + player.width/2;
bullets[b3].y = player.y + player.height/2
    - bullets[b3].height/2;
bullets[b3].y += 16;
}
break;

```

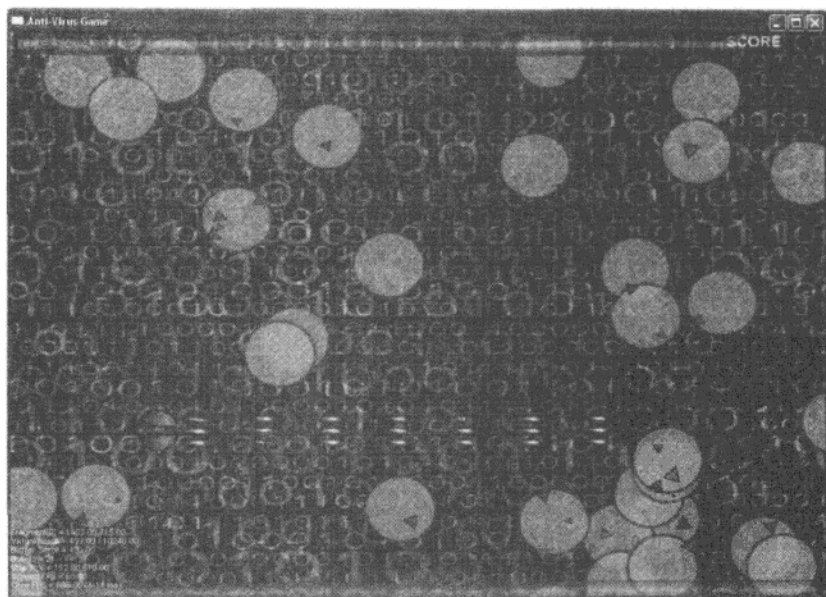


图 14-3 使用火力级别 3 一次发出三颗子弹

#### 4. 火力 4

第四级武器升级的特色是有四颗看起来很让人印象深刻的子弹，如图 14-4 所示。这是最后一个子弹的朝向相同的武器升级，在下一级我们将让子弹以不同方向发射。注意以下代码清单中

对四颗子弹中的每一颗独立进行定位并且“发射”的方法。即使我们看到四颗子弹以成组的方式一起出现，它们的移动及与环境的交互也是互相独立的。这是一个重要的游戏概念，希望读者能掌握。所以要学习代码！大多数游戏代码和这段代码一样，通过按需重复非常简单的代码来获得强大的能力。如果你想编写一个漂亮的算法或者通过某种方法将这四颗子弹放到循环中，建议你别做这样的事情！漂亮的算法并不意味着更快速的代码，这是由现代处理器架构的设计方法决定的：长的管线处理序列的能力要比算法好得多。

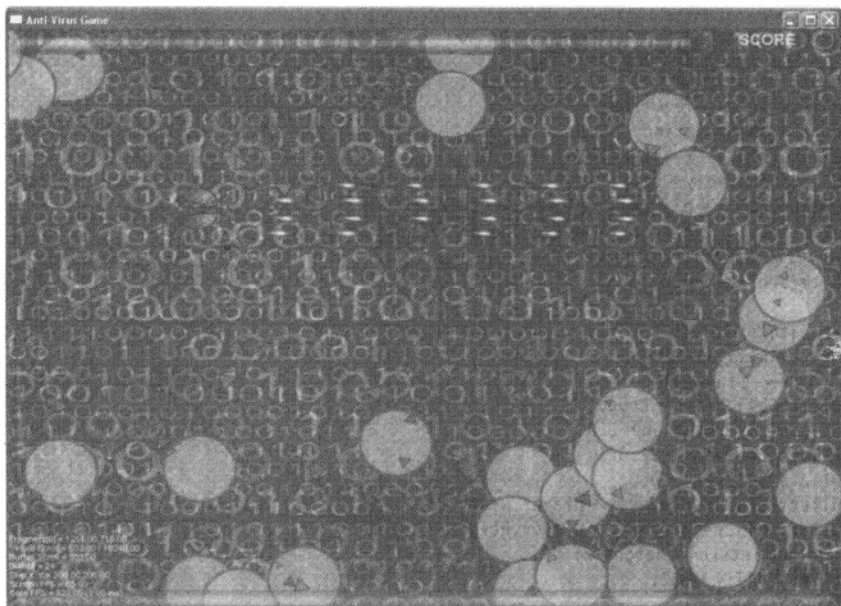


图 14-4 使用火力级别 4 一次发出四发子弹

```
case 4:
{
    //create bullet 1
    int b1 = find_bullet();
    if (b1 == -1) return;
    bullets[b1].alive = true;
    bullets[b1].rotation = 0.0;
    bullets[b1].velx = 12.0f;
    bullets[b1].vely = 0.0f;
    bullets[b1].x = player.x + player.width/2;
    bullets[b1].x += 8;
    bullets[b1].y = player.y + player.height/2
        - bullets[b1].height/2;
    bullets[b1].y -= 12;

    //create bullet 2
    int b2 = find_bullet();
    if (b2 == -1) return;
```



```

bullets[b2].alive = true;
bullets[b2].rotation = 0.0;
bullets[b2].velx = 12.0f;
bullets[b2].vely = 0.0f;
bullets[b2].x = player.x + player.width/2;
bullets[b2].x += 8;
bullets[b2].y = player.y + player.height/2
    - bullets[b2].height/2;
bullets[b2].y += 12;

//create bullet 3
int b3 = find_bullet();
if (b3 == -1) return;
bullets[b3].alive = true;
bullets[b3].rotation = 0.0;
bullets[b3].velx = 12.0f;
bullets[b3].vely = 0.0f;
bullets[b3].x = player.x + player.width/2;
bullets[b3].y = player.y + player.height/2
    - bullets[b3].height/2;
bullets[b3].y -= 32;

//create bullet 4
int b4 = find_bullet();
if (b4 == -1) return;
bullets[b4].alive = true;
bullets[b4].rotation = 0.0;
bullets[b4].velx = 12.0f;
bullets[b4].vely = 0.0f;
bullets[b4].x = player.x + player.width/2;
bullets[b4].y = player.y + player.height/2
    - bullets[b4].height/2;
bullets[b4].y += 32;
}
break;

```

## 5. 火力 5

第五级也是最后一级火力升级，这一级别和我们在前面四级中使用的模式不同。在这一级别中，我们将以从纳米机器人散开的角度来打出子弹，而不是直线向前。从此以后，读者就可以按最大的创意来处理火力——我会给出读者所需的代码并让读者想到一些有创意的新的可能性！见图 14-5。

在你把下列代码当成理所当然的东西之前，一定要更仔细一些，因为它与前面的火力代码有些不同！在这个情况下使用了两个新函数。下面将突出显示这些函数调用。

```

case 5:
{
    //create bullet 1
    int b1 = find_bullet();
    if (b1 == -1) return;
    bullets[b1].alive = true;
    bullets[b1].rotation = 0.0;
    bullets[b1].velx = 12.0f;

```

```

bullets[b1].vely = 0.0f;
bullets[b1].x = player.x + player.width/2;
bullets[b1].y = player.y + player.height/2
    - bullets[b1].height/2;
bullets[b1].y -= 12;

//create bullet 2
int b2 = find_bullet();
if (b2 == -1) return;
bullets[b2].alive = true;
bullets[b2].rotation = 0.0;
bullets[b2].velx = 12.0f;
bullets[b2].vely = 0.0f;
bullets[b2].x = player.x + player.width/2;
bullets[b2].y = player.y + player.height/2
    - bullets[b2].height/2;
bullets[b2].y += 12;

//create bullet 3
int b3 = find_bullet();
if (b3 == -1) return;
bullets[b3].alive = true;
bullets[b3].rotation = -4.0;
bullets[b3].velx = (float) (12.0 *
    LinearVelocityX( bullets[b3].rotation ));
bullets[b3].vely = (float) (12.0 *
    LinearVelocityY( bullets[b3].rotation ));
bullets[b3].x = player.x + player.width/2;
bullets[b3].y = player.y + player.height/2
    - bullets[b3].height/2;
bullets[b3].y -= 20;

//create bullet 4
int b4 = find_bullet();
if (b4 == -1) return;
bullets[b4].alive = true;
bullets[b4].rotation = 4.0;
bullets[b4].velx = (float) (12.0 *
    LinearVelocityX( bullets[b4].rotation ));
bullets[b4].vely = (float) (12.0 *
    LinearVelocityY( bullets[b4].rotation ));
bullets[b4].x = player.x + player.width/2;
bullets[b4].y = player.y + player.height/2
    - bullets[b4].height/2;
bullets[b4].y += 20;
}
break;

```

这里有两颗朝前方向的子弹和两颗偏离一个角度的子弹，这极大地增加了射击的威力，因为这种类型的火力覆盖了更多的屏幕空间！关键在于两个函数：LinearVelocityX() 和 LinearVelocityY()。



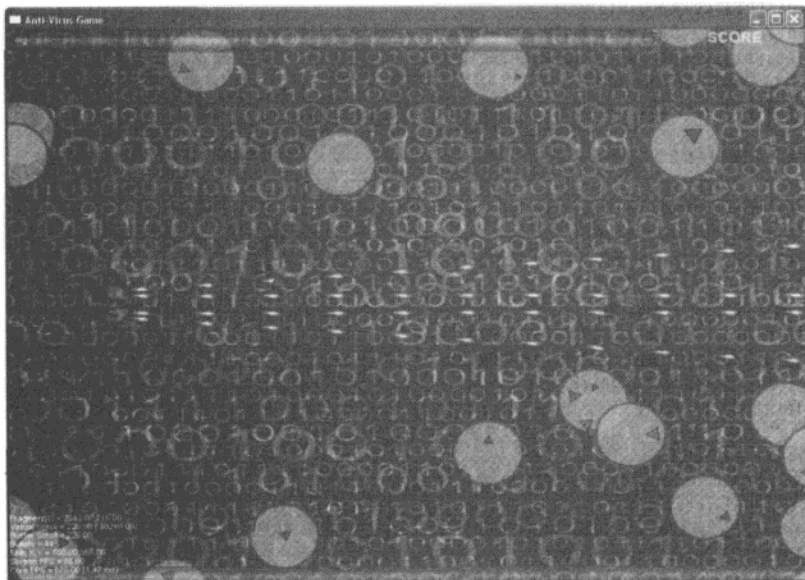


图 14-5 使用火力级别 5 发出子弹

```
const double PI = 3.1415926535;
const double PI_over_180 = PI / 180.0f;

double LinearVelocityX(double angle)
{
    if (angle < 0) angle = 360 + angle;
    return cos( angle * PI_over_180);
}

double LinearVelocityY(double angle)
{
    if (angle < 0) angle = 360 + angle;
    return sin( angle * PI_over_180);
}
```

在调用这些函数中的任意一个时，需要传递角度。这些函数会在内部将角度转换为弧度，因为  $\sin()$  和  $\cos()$ （分别计算正弦和余弦）只能处理弧度。将一定的角度传递给  $\text{LinearVelocityX}()$ ，它将计算精灵以这个方向移动所需的 X 速度。同样的，将一定的角度传递给  $\text{LinearVelocityY}()$ ，它将计算精灵的 Y 速度。将这两个值组合在一起，就可让精灵（例如子弹）以一定的方向在屏幕上移动。这些值通常很小，其范围在 0.0 ~ 1.0 之间，所以如果想让精灵移动得更快，可以乘以一个数。

**建议** 记得在笛卡尔坐标系中  $0^\circ$  是朝向右边而不是像罗盘那样朝上的！朝上的方向实际上是  $-90^\circ$ ，或者  $+270^\circ$ （如果顺时针转圈的话）。朝下是  $+90^\circ$ 。

## 6. 给纳米机器人充电

所有这些疯狂的火力当然是要有代价的，它会消耗大量能量。我们的小纳米机器人只能在其很小的储能器中保存一定的能量。所以，充电是必需的，而且得频繁充电！幸运的是，在计算机内存中有程序片段到处漂浮，我们可以消费之。按键盘上的 X 键或控制器的 X 按钮来启动充电系统，如图 14-6 所示。

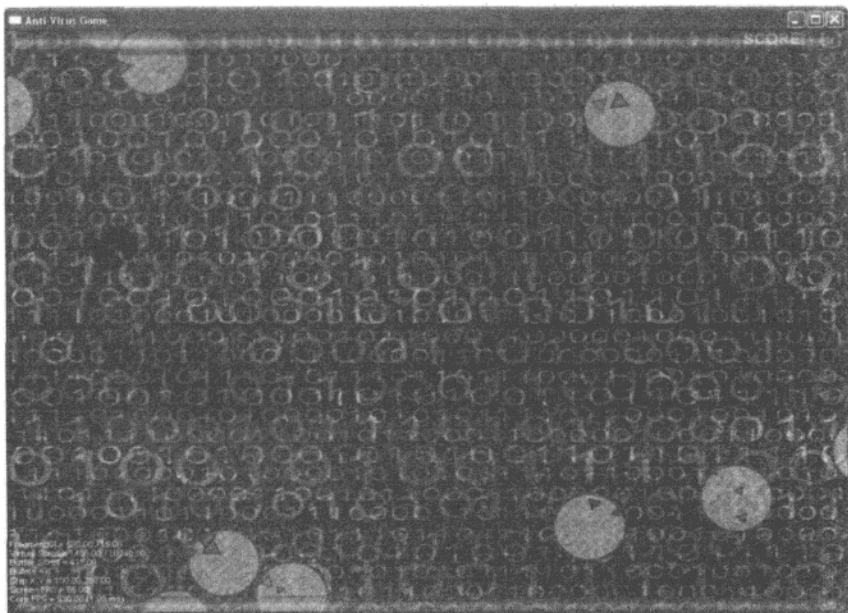


图 14-6 通过吸入几何形状的能力节点来给纳米机器人的储能器充电

## 7. 过载武器系统

除了正常的火力级别以外，RAINeR 纳米机器人可以过载其武器系统，从而一次产生大规模子弹爆炸，这很适合在有太多外星病毒临近时扫清道路！过载的顺序见图 14-7，图 14-8 显示了子弹炸开产生的能量风暴！由于以这种方式对武器系统进行过载的危险本质，故障保护系统将会在纳米机器人的能量储存到达 50% 时执行发射。所以，要想获得最有效的过载发射，需要先有一次完全的充电。

### 14.1.2 游戏源代码

以下是游戏的完整源代码。在本章中读者将看不到我们以前已经在 MyDirectX 等文件中给出的代码。即使如此，在没有任何框架代码的情况下我们也将看到很长的代码——1 100 行。而这只是原型或者伪游戏而已，不是真正的 Anti-Virus 游戏！在源代码清单中，以粗体突出了希望读者特别注意的重点代码行——因为这些代码行是读者在短时间内可以给游戏做出最重要的更改的

地方。而从长期来说，我们需看看这个游戏最终会是什么样。你会是完成这一游戏的人吗？

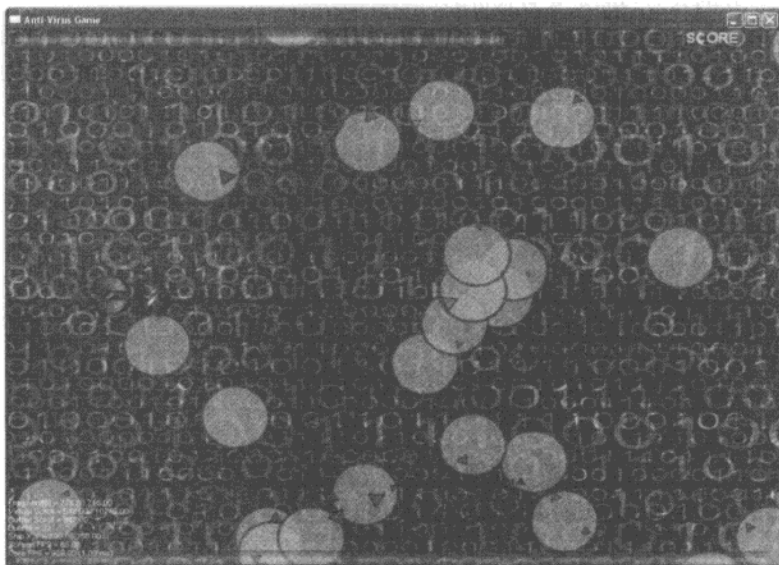


图 14-7 准备过载发射

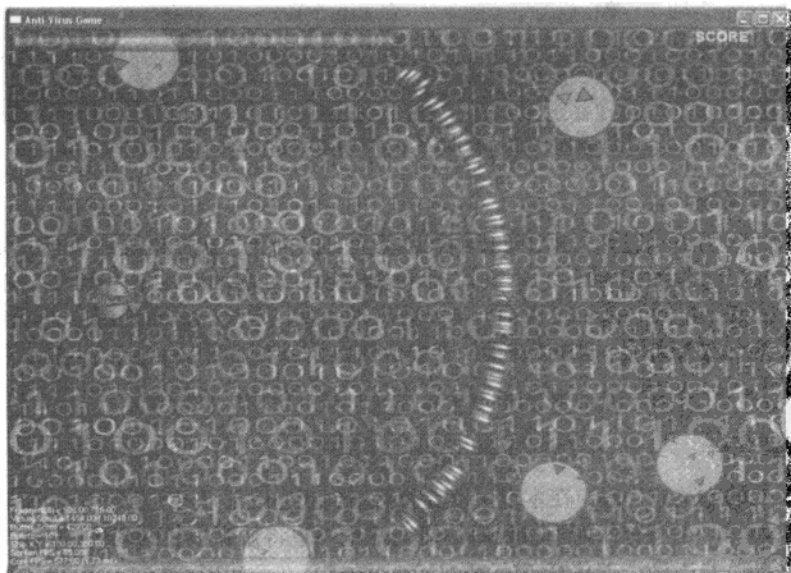


图 14-8 发射过载子弹可消灭视线内的几乎所有东西

```
/*
    Beginning Game Programming, Third Edition
    Anti-Virus Game
    MyGame.cpp
*/

#include "MyDirectX.h"
#include <sstream>
using namespace std;

const string APPTITLE = "Anti-Virus Game";
const int SCREENW = 1024;
const int SCREENH = 768;

//font variables
LPD3DXFONT font;
LPD3DXFONT debugfont;

//timing variables
bool paused = false;
DWORD refresh = 0;
DWORD screentime = 0;
double screenfps = 0.0;
double screencount = 0.0;
DWORD coretime = 0;
double corefps = 0.0;
double corecount = 0.0;
DWORD currenttime;

//background scrolling variables
const int BUFFERW = SCREENW * 2;
const int BUFFERH = SCREENH;
double scrollx=0;
double scrolly=0;
LPDIRECT3DSURFACE9 background = NULL;
const double virtual_level_size = BUFFERW * 5;
double virtual_scrollx = 0;

//player variables
LPDIRECT3DTEXTURE9 player_ship;
SPRITE player;
enum PLAYER_STATES
{
    NORMAL = 0,
    PHASING = 1,
    CHARGING = 2,
    OVERLOADING = 3
};
PLAYER_STATES player_state = NORMAL;
PLAYER_STATES player_state_previous = NORMAL;
D3DXVECTOR2 position_history[8];
int position_history_index = 0;
DWORD position_history_timer = 0;
double charge_angle = 0.0;
```



```
double charge_tweak = 0.0;
double charge_tweak_dir = 1.0;
double energy = 100.0;
double health = 100.0;

//enemy virus objects
const int VIRUSES = 100;
LPDIRECT3DTEXTURE9 virus1_image;
SPRITE viruses[VIRUSES];

const int FRAGMENTS = 300;
LPDIRECT3DTEXTURE9 fragment_image;
SPRITE fragments[FRAGMENTS];

//bullet variables
LPDIRECT3DTEXTURE9 purple_fire;
const int BULLETS = 300;
SPRITE bullets[BULLETS];
int player_shoot_timer = 0;
int firepower = 1;
int bulletcount = 0;

//sound effects
CSound *tisk=NULL;
CSound *foom=NULL;
CSound *charging=NULL;

//GUI elements
LPDIRECT3DTEXTURE9 energy_slice;
LPDIRECT3DTEXTURE9 health_slice;
//controller vibration
int vibrating = 0;
int vibration = 100;

/**
** Game initialization
**/
bool Game_Init(HWND window)
{
    Direct3D_Init(window, SCREENW, SCREENH, false);
    DirectInput_Init(window);
    DirectSound_Init(window);

    //create a font
    font = MakeFont("Arial Bold", 24);
    debugfont = MakeFont("Arial", 14);

    //load background
    LPDIRECT3DSURFACE9 image = NULL;
    image = LoadSurface("binary.bmp");
    if (!image) return false;
```



```

//load player sprite
player_ship = LoadTexture("ufo.png");
player.x = 100;
player.y = 350;
player.width = player.height = 64;

for (int n=0; n<4; n++)
    position_history[n] = D3DXVECTOR2(-100,0);

//load bullets
purple_fire = LoadTexture("purplefire.tga");
for (int n=0; n<BULLETS; n++)
{
    bullets[n].alive = false;
    bullets[n].x = 0;
    bullets[n].y = 0;
    bullets[n].width = 55;
    bullets[n].height = 16;
}

//load enemy viruses
virus1_image = LoadTexture("virus1.tga");
for (int n=0; n<VIRUSES; n++)
{
    viruses[n].alive = true;
    viruses[n].width = 96;
    viruses[n].height = 96;
    viruses[n].x = (float)(1000 + rand() % BUFFERW);
    viruses[n].y = (float)(rand() % SCREENH);
    viruses[n].velx = (float)((rand() % 8) * -1);
    viruses[n].vely = (float)(rand() % 2 - 1);
}

//load gui elements
energy_slice = LoadTexture("energyslice.tga");
health_slice = LoadTexture("healthslice.tga");

//load audio files
tisk = LoadSound("clip.wav");
foom = LoadSound("longfoom.wav");
charging = LoadSound("charging.wav");

//load memory fragments (energy)
fragment_image = LoadTexture("fragment.tga");
for (int n=0; n<FRAGMENTS; n++)
{
    fragments[n].alive = true;
    D3DCOLOR fragmentcolor = D3DCOLOR_ARGB(
        200 + rand() % 55,
        150 + rand() % 100,
        150 + rand() % 100,
        150 + rand() % 100);
}

```



```
fragments[n].color = fragmentcolor;
fragments[n].width = 128;
fragments[n].height = 128;
fragments[n].scaling = (float)(rand() % 20 + 10) / 150.0f;
fragments[n].rotation = (float)( rand() % 360 );
fragments[n].velx = (float)(rand() % 4 + 1) * -1.0f;
fragments[n].vely = (float)(rand() % 10 - 5) / 10.0f;
fragments[n].x = (float)( rand() % BUFFERW );
fragments[n].y = (float)( rand() % SCREENH );
}
```

```
//create background for scroller
HRESULT result =
d3ddev->CreateOffscreenPlainSurface(
    BUFFERW,
    BUFFERH,
    D3DFMT_X8R8G8B8,
    D3DPPOOL_DEFAULT,
    &background,
    NULL);
if (result != D3D_OK) return false;

//copy image to upper left corner of background
RECT source_rect = {0, 0, SCREENW, SCREENH };
RECT dest_ul = { 0, 0, SCREENW, SCREENH };

d3ddev->StretchRect(
    image,
    &source_rect,
    background,
    &dest_ul,
    D3DTEXF_NONE);

//copy image to upper right corner of background
RECT dest_ur = { SCREENW, 0, SCREENW*2, SCREENH };

d3ddev->StretchRect(
    image,
    &source_rect,
    background,
    &dest_ur,
    D3DTEXF_NONE);

//get pointer to the back buffer
d3ddev->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO,
    &backbuffer);

//remove scratch image
image->Release();
```



```
        return true;
    }

    /**
     ** Game shutdown
     **/
    void Game_End()
    {
        background->Release();
        font->Release();
        debugfont->Release();
        fragment_image->Release();
        delete charging;
        delete foom;
        delete tisk;
        energy_slice->Release();
        health_slice->Release();
        virus1_image->Release();
        purple_fire->Release();
        player_ship->Release();

        DirectSound_Shutdown();
        DirectInput_Shutdown();
        Direct3D_Shutdown();
    }

    /**
     ** Helper function used to print out variables
     **/
    template <class T>
    std::string static ToString(const T &t, int places = 2)
    {
        ostringstream oss;
        oss.precision(places);
        oss.setf(ios_base::fixed);
        oss << t;
        return oss.str();
    }

    /**
     ** Updates the player's position
     **/
    void move_player(float movex, float movey)
    {
        //cannot move while overloading!
        if (player_state == OVERLOADING
            || player_state_previous == OVERLOADING)
            return;

        float multi = 4.0f;
        player.x += movex * multi;
        player.y += movey * multi;
    }
```





```
/**
** Math helper functions
**/
const double PI = 3.1415926535;
const double PI_under_180 = 180.0f / PI;
const double PI_over_180 = PI / 180.0f;

double toRadians(double degrees)
{
    return degrees * PI_over_180;
}

double toDegrees(double radians)
{
    return radians * PI_under_180;
}

double wrap(double value, double bounds)
{
    double result = fmod(value, bounds);
    if (result < 0) result += bounds;
    return result;
}

double wrapAngleDegs(double degs)
{
    return wrap(degs, 360.0);
}

double LinearVelocityX(double angle)
{
    //angle -= 90;
    if (angle < 0) angle = 360 + angle;
    return cos( angle * PI_over_180);
}

double LinearVelocityY(double angle)
{
    //angle -= 90;
    if (angle < 0) angle = 360 + angle;
    return sin( angle * PI_over_180);
}

/**
** Increases the energy capacitor level
**/
void add_energy(double value)
{
    energy += value;
    if (energy < 0.0) energy = 0.0;
    if (energy > 100.0) energy = 100.0;
}

/**
```

```

** Handles vibrating the controller; most importantly,
** this function also helps to stop the vibration
**/

```

```

void Vibrate(int contnum, int amount, int length)

```

```

{
    vibrating = 1;
    vibration = length;
    XInput_Vibrate(contnum, amount);
}

```

```

/**
** Look for an unused bullet in the array
**/

```

```

int find_bullet()

```

```

{
    int bullet = -1;
    for (int n=0; n<BULLETS; n++)
    {
        if (!bullets[n].alive)
        {
            bullet = n;
            break;
        }
    }
    return bullet;
}

```

```

/**
** Add a bullet to the overload group
**/

```

```

bool player_overload()

```

```

{
    //disallow overload unless energy is at 100%
    if (energy < 50.0) return false;

    //reduce energy for this shot
    add_energy(-0.5);
    if (energy < 0.0)
    {
        energy = 0.0;
        return false;
    }

    //play charging sound
    PlaySound(charging);

    //vibrate controller
    Vibrate(0, 20000, 20);

    int b1 = find_bullet();
    if (b1 == -1) return true;
    bullets[b1].alive = true;
    bullets[b1].velx = 0.0f;
}

```



```
    bullets[b1].vely = 0.0f;
    bullets[b1].rotation = (float)(rand() % 360);
    bullets[b1].x = player.x + player.width;
    bullets[b1].y = player.y + player.height/2
        - bullets[b1].height/2;
    bullets[b1].y += (float)(rand() % 20 - 10);

    return true;
}
/**
** Fire a shot with firepower upgrades
**/
void player_shoot()
{
    //limit firing rate
    if ((int)timeGetTime() < player_shoot_timer + 100) return;
    player_shoot_timer = timeGetTime();

    //reduce energy for this shot
    add_energy(-1.0);
    if (energy < 1.0)
    {
        energy = 0.0;
        return;
    }

    //play firing sound
    PlaySound(tisk);

    Vibrate(0, 25000, 10);

    //launch bullets based on firepower level
    switch(firepower)
    {
        case 1:
        {
            //create a bullet
            int b1 = find_bullet();
            if (b1 == -1) return;
            bullets[b1].alive = true;
            bullets[b1].rotation = 0.0;
            bullets[b1].velx = 12.0f;
            bullets[b1].vely = 0.0f;
            bullets[b1].x = player.x + player.width/2;
            bullets[b1].y = player.y + player.height/2
                - bullets[b1].height/2;
        }
        break;
        case 2:
        {
            //create bullet 1
            int b1 = find_bullet();
```

```
if (b1 == -1) return;
bullets[b1].alive = true;
bullets[b1].rotation = 0.0;
bullets[b1].velx = 12.0f;
bullets[b1].vely = 0.0f;
bullets[b1].x = player.x + player.width/2;
bullets[b1].y = player.y + player.height/2
    - bullets[b1].height/2;
bullets[b1].y -= 10;

//create bullet 2
int b2 = find_bullet();
if (b2 == -1) return;
bullets[b2].alive = true;
bullets[b2].rotation = 0.0;
bullets[b2].velx = 12.0f;
bullets[b2].vely = 0.0f;
bullets[b2].x = player.x + player.width/2;
bullets[b2].y = player.y + player.height/2
    - bullets[b2].height/2;
bullets[b2].y += 10;
}
break;

case 3:
{
    //create bullet 1
    int b1 = find_bullet();
    if (b1 == -1) return;
    bullets[b1].alive = true;
    bullets[b1].rotation = 0.0;
    bullets[b1].velx = 12.0f;
    bullets[b1].vely = 0.0f;
    bullets[b1].x = player.x + player.width/2;
    bullets[b1].y = player.y + player.height/2
        - bullets[b1].height/2;

    //create bullet 2
    int b2 = find_bullet();
    if (b2 == -1) return;
    bullets[b2].alive = true;
    bullets[b2].rotation = 0.0;
    bullets[b2].velx = 12.0f;
    bullets[b2].vely = 0.0f;
    bullets[b2].x = player.x + player.width/2;
    bullets[b2].y = player.y + player.height/2
        - bullets[b2].height/2;
    bullets[b2].y -= 16;

    //create bullet 3
    int b3 = find_bullet();
    if (b3 == -1) return;
    bullets[b3].alive = true;
    bullets[b3].rotation = 0.0;
```



```
bullets[b3].velx = 12.0f;
bullets[b3].vely = 0.0f;
bullets[b3].x = player.x + player.width/2;
bullets[b3].y = player.y + player.height/2
    - bullets[b3].height/2;
bullets[b3].y += 16;

}
break;

case 4:
{
    //create bullet 1
    int b1 = find_bullet();
    if (b1 == -1) return;
    bullets[b1].alive = true;
    bullets[b1].rotation = 0.0;
    bullets[b1].velx = 12.0f;
    bullets[b1].vely = 0.0f;
    bullets[b1].x = player.x + player.width/2;
    bullets[b1].x += 8;
    bullets[b1].y = player.y + player.height/2
        - bullets[b1].height/2;
    bullets[b1].y -= 12;

    //create bullet 2
    int b2 = find_bullet();
    if (b2 == -1) return;
    bullets[b2].alive = true;
    bullets[b2].rotation = 0.0;
    bullets[b2].velx = 12.0f;
    bullets[b2].vely = 0.0f;
    bullets[b2].x = player.x + player.width/2;
    bullets[b2].x += 8;
    bullets[b2].y = player.y + player.height/2
        - bullets[b2].height/2;
    bullets[b2].y += 12;

    //create bullet 3
    int b3 = find_bullet();
    if (b3 == -1) return;
    bullets[b3].alive = true;
    bullets[b3].rotation = 0.0;
    bullets[b3].velx = 12.0f;
    bullets[b3].vely = 0.0f;
    bullets[b3].x = player.x + player.width/2;
    bullets[b3].y = player.y + player.height/2
        - bullets[b3].height/2;
    bullets[b3].y -= 32;

    //create bullet 4
    int b4 = find_bullet();
    if (b4 == -1) return;
    bullets[b4].alive = true;
```



```
bullets[b4].rotation = 0.0;
bullets[b4].velx = 12.0f;
bullets[b4].vely = 0.0f;
bullets[b4].x = player.x + player.width/2;
bullets[b4].y = player.y + player.height/2
    - bullets[b4].height/2;
bullets[b4].y += 32;
}
break;

case 5:
{
    //create bullet 1
    int b1 = find_bullet();
    if (b1 == -1) return;
    bullets[b1].alive = true;
    bullets[b1].rotation = 0.0;
    bullets[b1].velx = 12.0f;
    bullets[b1].vely = 0.0f;
    bullets[b1].x = player.x + player.width/2;
    bullets[b1].y = player.y + player.height/2
        - bullets[b1].height/2;
    bullets[b1].y -= 12;
    //create bullet 2
    int b2 = find_bullet();
    if (b2 == -1) return;
    bullets[b2].alive = true;
    bullets[b2].rotation = 0.0;
    bullets[b2].velx = 12.0f;
    bullets[b2].vely = 0.0f;
    bullets[b2].x = player.x + player.width/2;
    bullets[b2].y = player.y + player.height/2
        - bullets[b2].height/2;
    bullets[b2].y += 12;

    //create bullet 3
    int b3 = find_bullet();
    if (b3 == -1) return;
    bullets[b3].alive = true;
    bullets[b3].rotation = -4.0; // 86.0;
    bullets[b3].velx = (float) (12.0 *
        LinearVelocityX( bullets[b3].rotation ));
    bullets[b3].vely = (float) (12.0 *
        LinearVelocityY( bullets[b3].rotation ));
    bullets[b3].x = player.x + player.width/2;
    bullets[b3].y = player.y + player.height/2
        - bullets[b3].height/2;
    bullets[b3].y -= 20;

    //create bullet 4
    int b4 = find_bullet();
    if (b4 == -1) return;
    bullets[b4].alive = true;
```

```

        bullets[b4].rotation = 4.0; // 94.0;
        bullets[b4].velx = (float) (12.0 *
            LinearVelocityX( bullets[b4].rotation ));
        bullets[b4].vely = (float) (12.0 *
            LinearVelocityY( bullets[b4].rotation ));
        bullets[b4].x = player.x + player.width/2;
        bullets[b4].y = player.y + player.height/2
            - bullets[b4].height/2;
        bullets[b4].y += 20;
    }
    break;
}
}
}

```

由于程序太长，所以删节了 Anti-Virus 游戏源代码的剩余部分，完整的代码可以在 CD-ROM 中找到。打开本项目，就可以查看剩下的源代码。如果没有 CD-ROM，访问 [www.jharbour.com/forum](http://www.jharbour.com/forum) 下载该项目。

## 14.2 你所学到的

这就结束了 Anti-Virus 游戏的介绍。这个游戏有巨大的潜力！真正的问题不是这个伪游戏能做什么，而是有什么是它不能做的！我们有一个很棒的故事线索，而且已经有一些非常好的游戏玩法（只需再做一些设计），还有功能齐备且已经能工作的开火代码。这是一个单向卷动的射击游戏，有才华且愿意在它上面花时间的人，大可把它变成一个伟大的游戏！这个人会是你吗？

## 14.3 复习测验

以下复习测验题将考查你是否掌握了本章的所有内容。

- 1) 本游戏当前有多少个级别的火力？
- 2) 计算以给定角度为基础的精灵的 X 速度时调用的三角函数是哪一个？
- 3) 计算以给定角度为基础的精灵的 Y 速度时调用的三角函数是哪一个？
- 4) 这个游戏用于卷动背景的 Direct3D 对象是什么类型：表面还是纹理？
- 5) 按纳米机器人的火力级别处理发射子弹的函数是哪一个？
- 6) 游戏中的程序片段是什么形状的？
- 7) 游戏中程序片段是做什么用的？
- 8) 在游戏中，敌方病毒是否攻击游戏者？为什么？
- 9) 简要解释一下游戏同时处理所有要渲染的精灵的方法。
- 10) 字体打印函数的名称是什么？

## 14.4 自己动手

我只在这里对 Anti-Virus 游戏给出两个建议，虽然我很容易就能很快地在脑海中蹦出 30 个来。

习题 1 Anti-Virus 游戏的特点是敌方病毒对象似乎不以任何方式影响游戏者的纳米机器人。修改程序, 让敌方病毒精灵对游戏者有危险, 只要发生碰撞就会破坏或毁灭纳米机器人。

习题 2 进一步修改 Anti-Virus 游戏, 让敌方病毒爆炸成更小的物件, 每一块都对游戏者的纳米机器人产生危险, 都需要躲避或毁灭它。







## 第四部分 >>>

### 附 录

---

本附录将提供正文中没有涵盖的（为了改进行文和可读性）或者作为正文补充的（比如章节测验答案）资料。附录 D 附加了一些我本想在正文中包含但却因为时间限制或某些主题的难度问题而不得不删去的示例。本书一共有 4 个附录：

- 附录 A：配置 Visual C++
- 附录 B：可进一步学习的资源
- 附录 C：各章测验答案
- 附录 D：附加示例



## 附录 A 配置 Visual C++

为简化起见，本书采用了更为一般的方法来配置编译器项目。因为普通的初学者（按照我对第一次接触 DirectX 的学生的经验）难以理解各个文件到底属于哪个地方。而且 Visual C++ 2008 是一个复杂的开发环境，有许多许多的选项。这对初学者而言显得有些可怕，即使是对 C++ 语言已经很有经验的人也会有所惧怕。

为了让项目的配置简单，首先把项目的所有“资产”（位图文件、音频文件等）存储在项目文件夹中。目前，项目文件夹和解决方案文件夹是同样的。解决方案文件以 .SLN 为扩展名，在 Visual C++ 2008 中创建新项目时它通常在主文件夹中。包含项目文件（扩展名为 .VCPROJ）的项目文件夹将会是解决方案文件夹之下的一个子文件夹。将游戏的所有资产放到项目文件夹，而后再在 Visual C++ 中调试时（用 F5 键），它就能找到这些资产。

### A.1 手工设置文件夹

在安装 DirectX 之前必须先安装 Visual C++ 2008，这很重要。从 Microsoft 可以下载两个版本的 DirectX：runtime（运行时）和 software development kit（软件开发包，也就是 SDK）。我们需要安装 DirectX SDK，它也包含了 runtime（运行时）。如果只安装 runtime（运行时）（大多数新游戏都带它），那么就无法编译本书中的任何 DirectX 代码。确认先安装 Visual C++ 2008，然后到 <http://msdn.microsoft.com/directx/> 下载并安装 DirectX SDK（本书编写时的最新版本是 2009 年 3 月推出的）。确认在安装 Visual C++ 之后安装 DirectX，以便能正确配置。

如果先安装 DirectX，没问题，在安装 Visual C++ 之后重新安装 DirectX SDK 即可。这么做是必需的，因为 DirectX 安装程序将自动配置 Visual C++，告诉它在编译 DirectX 代码时应该在哪儿找到 DirectX SDK 文件。否则，用户就必须手工添加 DirectX 文件夹。无论因为什么原因，如果万一需要这么做的话，可通过打开 Tools 菜单，选择 Options，然后选择 Projects and Solutions，然后选择 VC++ Directories 来配置文件夹。在 Show directories for 下拉列表中选择 Include files，然后添加 DirectX 包含文件夹（通常在 C:\Program Files\Microsoft DirectX SDK(Month Year) 文件夹中）。对 Library files 项重复同样的操作，加入对 DirectX SDK\Lib\x86 文件夹的引用。

### A.2 创建新项目

首先，我们来看创建新项目的方法。打开 Visual C++ 2008。如果尚未下载安装，那么跳到 <http://www.microsoft.com/Express/vc/>，然后下载、安装 Visual C++ 2008。这是很容易做的事情！在运行 Visual C++ 2008 时，将显示集成开发环境（IDE），如图 A-1 所示。

打开 File 菜单，选择 New → Project。系统显示 New Project 对话框，如图 A-2 所示。在本例中，我们使用的是 Professional 版，该版要比读者可能使用的 Express 版有更多项目类型。打开 Visual C++ 列表项，选择 Win32。（通常情况下）读者将看到两种 Win32 项目模板，一种是控制台项目，另一种

是“Win32 项目”。选择后一种作为 DirectX 项目。键入新项目的名称并选择一个位置。

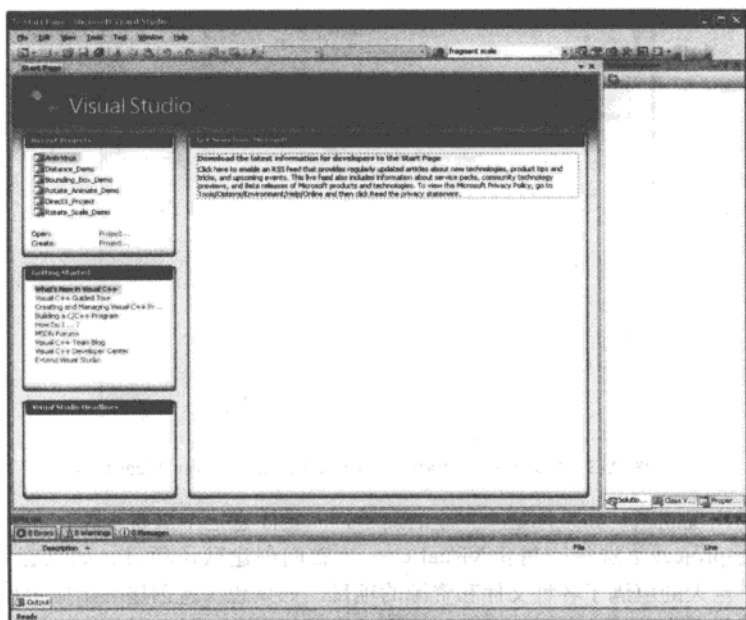


图 A-1 Visual C++ 2008 IDE

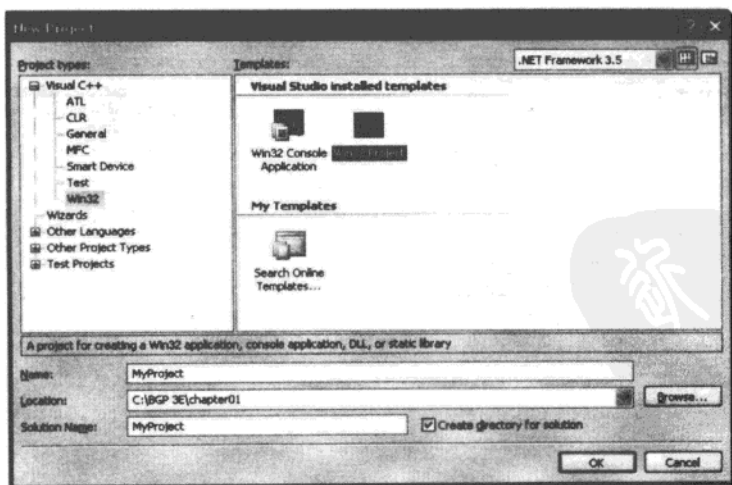


图 A-2 Visual C++ 2008 中的 New Project 对话框

下一步出现 Win32 Application Wizard 对话框，如图 A-3 所示。在左边的 Overview 页面链接之下选择 Application Settings。系统默认选择的是 Windows application 类型。

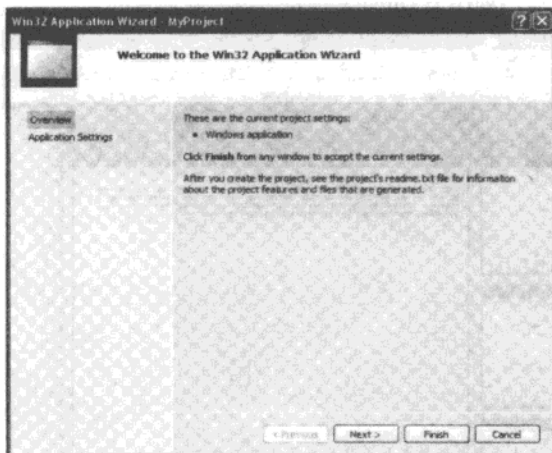


图 A-3 Visual C++ 2008 中的 Win32 Application Wizard

这里很容易出现问题！如果你是个完全的新手，一定注意这个重要步骤！

选中 Empty project 复选框，禁止 Visual C++ 为我们创建默认项目。如果忘了选这个选项，系统将生成一个巨大的填满了各种文件和资源的项目。如果想快速创建一个应用程序，那么这个选项很有帮助，但对于 DirectX 程序，我们不需要任何这些应用程序功能。如果不小心让 Empty project 选项处于未选中状态（不选中），那么就会得到一个如图 A-4 所示的项目。

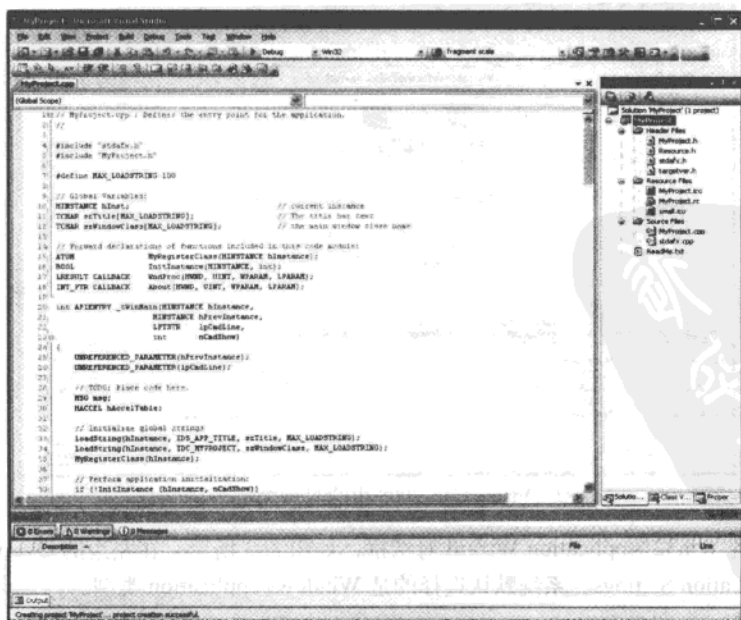


图 A-4 Win32 Application Wizard 生成的一个大的示例项目（这是我们不需要的）

通过选中 Empty project 选项, 我们会得到一个合适的项目。在有了一个整洁但空白的项目之后, 我们需要对其添加源代码文件。如图 A-5 所示, 打开 Project 菜单并选择 Add New Item。系统弹出 Add New Item 对话框, 如图 A-6 所示。

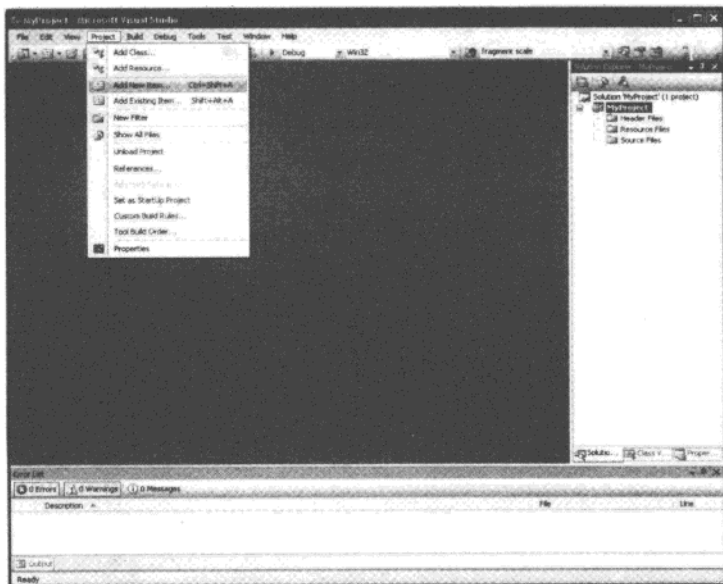


图 A-5 通过 Project 菜单添加新源代码文件

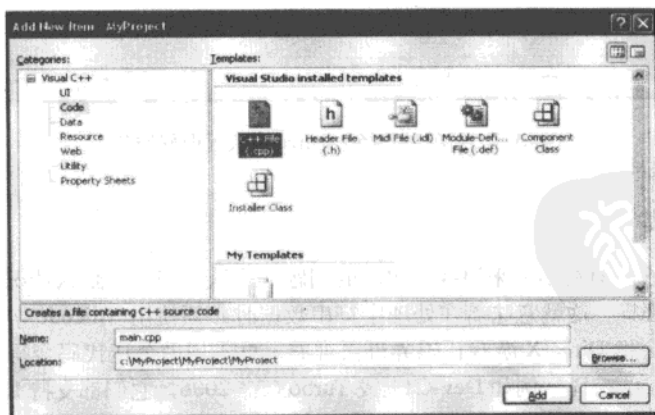


图 A-6 使用 Add New Item 对话框将新的 C++ 文件添加到项目中

在这个对话框中选择 C++ File(.cpp) 作为文件类型, 然后在 Name 字段中输入文件名。在该示例中输入的是 main.cpp。

在将新的 main.cpp 文件添加到项目之后, 整个开发环境应该和图 A-7 所示的相似。由于书

中所有代码都使用 `#pragma` 语句来给项目添加库文件（例如 `d3d9.lib`），所以无需再做任何配置上的改变了，DirectX 代码已经可以编译！这样就做好了！

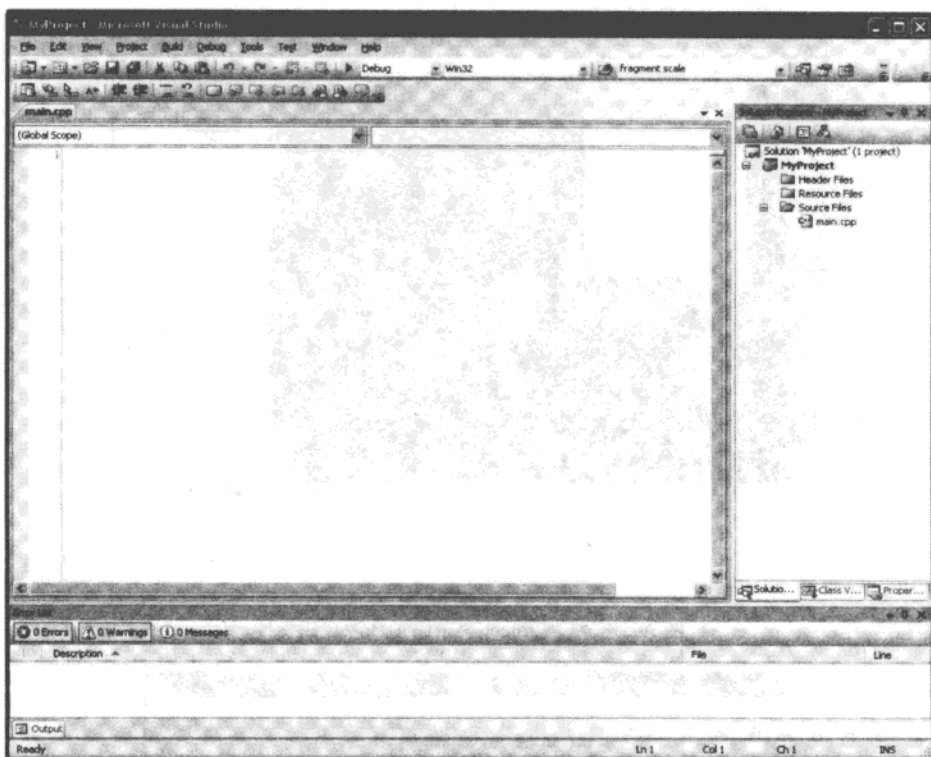


图 A-7 新项目已经为 DirectX 代码做好准备

### A.3 更改字符集

这里需要讲解一个对初学者来说经常出现的问题：默认字符集。默认情况下，新项目被配置为使用 Unicode 字符集，这就意味着在处理字符串数据时必须使用 Unicode 字符串转换函数。这很让人痛苦，而且会把 DirectX 游戏代码弄得很难看，更不用说会让代码变得不标准，而且很可能在其他编译器中无法编译（例如 Dev-C++ 或 Turbo C++ 2006，它们都支持 DirectX 9）。

更改字符集是很容易的事情。打开 Project 菜单，选择底部的 Project Properties 菜单项。接下来会出现 Project Property Pages 对话框（见图 A-8）。打开 Configuration Properties 列表项目，然后选择 General 选项。在 General 属性页靠近底部的位置是 Character Set 属性。将其更改为 multi-byte（多字节）。问题解决了！

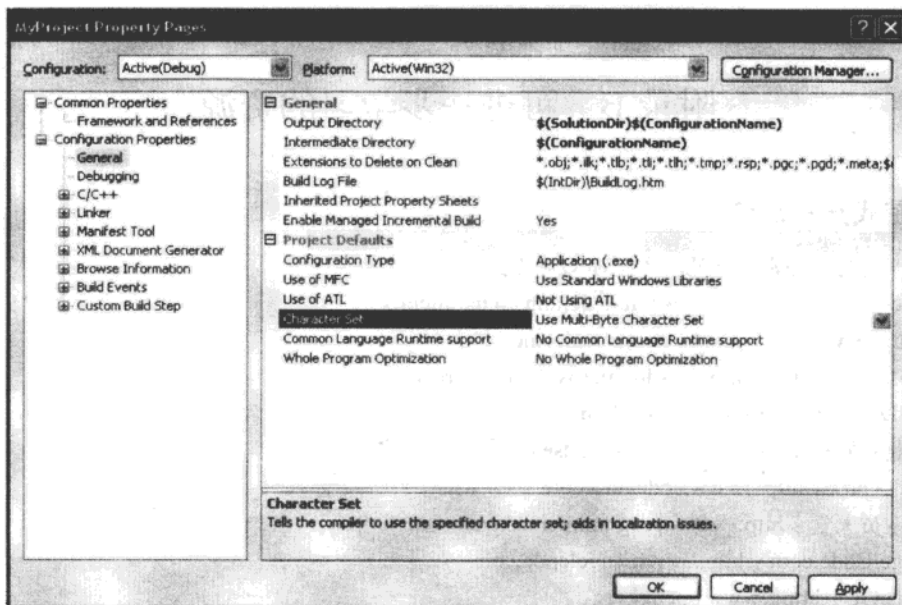


图 A-8 将默认字符集更改为 multi-byte



## 附录 B 可进一步学习的资源

### B.1 游戏开发站点

以下是笔者经常访问的优秀的游戏开发站点：

Allegro 主站：<http://www.talula.demon.co.uk/allegro/>

GameDev LCC：<http://www.gamedev.net>

MSDN DirectX：<http://msdn.microsoft.com/directx>

MSDN Visual C++：<http://msdn.microsoft.com/visualc>

游戏开发搜索引擎：<http://www.gdse.com>

CodeGuru：<http://www.codeguru.com>

程序员天堂：<http://www.programmersheaven.com>

AngelCode.com：<http://www.angelcode.com>

OpenGL：<http://www.opengl.org>

NeHe Productions：<http://nehe.gamedev.net>

NeXe：<http://nexe.gamedev.net>

游戏学院：<http://www.gameinstitute.com>

Wotsit 格式：<http://www.wotsit.org>

### B.2 出版物、游戏评论及下载站点

跟上最新潮流怎么说都是件可怕的任务。每分钟都有新的东西在这个世界上产生，希望下面这组链接能帮助读者跟上时代：

Course Technology：<http://www.courseptr.com>

Games Domain：<http://www.gamesdomain.com>

Blue's News：<http://www.bluesnews.com>

Download.com：<http://www.download.com>

Tucows：<http://www.tucows.com>

Slashdot：<http://slashdot.org>

Imagine Games Network (IGN)：<http://www.ign.com>

### B.3 行业

如果想进入这一行，就得知道这一行。阅读杂志，参加协会会议肯定能对读者有所帮助。

游戏开发者杂志：<http://www.gdmag.com>

GamaSutra : <http://www.gamasutra.com>  
 国际游戏开发者协会 : <http://www.igda.com>  
 游戏开发者会议 : <http://www.gdconf.com>  
 共享软件专业人士协会 : <http://www.asp-shareware.org>  
 RealGames : <http://www.real.com/games>

## B.4 计算机幽默

要是想找笑话，以下这些站点都很棒：

Homestar Runner (超级棒!) : <http://www.homestarrunner.com/>  
 用户友好 : <http://www.userfriendly.org>  
 土包子! : <http://www.happychaos.com/geeks>  
 不相关 : <http://www.offthemark.com/computers.htm>  
 游戏者对游戏者 : <http://www.pvponline.com>

## B.5 书籍推荐

在这里列出的每一本书都给出了一段简短的描述，因为它们要么是笔者写的（推销！）要么是笔者高度推荐的，认为它们在许多时候可以起到作用，是比较放松、有趣或者重要的书籍。读者会发现这里推荐的书籍不仅可以作为 C 语言的参考，同时也是本书讨论的主题的补充和参考。

《Advanced 2D Game Development》

Jonathan Harbour 著, Course Technology, ISBN 1598633422

这是笔者最新的书，是读完了入门书籍之后的读者很好的后续书籍。这是本为中级程序员写的书，让高级 2D 从头开始进入游戏引擎设计中，并且本书详细讲解了创建 2D 游戏引擎的方法。本书涵盖了 2D 游戏中最重要的内容，但它不是一本“游戏项目”书，而是一本引擎书，它可以帮助读者在 DirectX 的经验上更进一步。

《Introduction to 3D Game Programming with DirectX 9.0c: A Shader Approach》

Frank Luna 著, Wordware, ISBN 1598220160

这是一本全面介绍使用 Direct3D 9 进行 3D 图形编程的书籍，只不过不是为初学者所写，所以要想让本书对读者有益，读者需要有坚实的 C++ 基础并且对 DirectX 至少有实际使用经验。作为一本中高级水平的书，它将为读者讲授在当今游戏项目中所用的最为有用的技术，例如 3D 地形、角色动画及着色器编程。

《Programming an RTS Game with Direct3D》

Carl Granberg 著, Charles River, ISBN 1584504986

本书中开发的那个实时战略游戏是这方面题材的非凡示例，而且是任何中高级 DirectX 程序员的一个很好的学习工具。关于 3D 地形和动画的章节是必不可少的。虽然着色器代码是以 HLSL 2.0 文件格式来呈现的，但着色器程序仍旧可以编译和运行，而且很容易就可将其复制到 .fx 文件以便支持 3.0 版。

## 《Character Animation with Direct3D》

Carl Granberg 著, Charles River, ISBN 1584505702

这就是那本讲授 Direct3D 的 3D 网格动画技术的书,目前是市面上唯一一本最新潮的书(例如,有能工作的源代码示例)。笔者高度推荐它!需要预先告知的是,这是一本极为高级的书,在阅读它之前至少应该要读过 Granberg 的 RTS 书(或其他中级 DirectX 书籍)。

## 《AI Techniques for Game Programming》

Mat Buckland 著, Premier Press, ISBN 193184108X

“这本书涉及遗传算法和神经网络这些复杂的内容,但全书以平实的语言来讲解它们。书中没有其他书中常见的折磨人的数学公式和抽象的示例。每一章都是一步一步地带你进入理论之中,清晰地讲解了将每一项技术集成到读者自己的游戏中的方法”。

## 《DarkBASIC Pro Game Programming》,第2版

Jonathan S. Harbour 和 Joshua R. Smith 著, Premier Press, ISBN 1598632876

本书很好地介绍了使用 C 语言进行 Direct3D 编程、3D 图形组件编程的方法。

## 《Beginning C++ Game Programming》

Michael Dawson 著, Premier Press, ISBN 1592002056

“如果你做好了跳入游戏编程的世界的准备,那么本书将带你开始这趟旅行,并为你提供专业的游戏编程语言的坚实基础。读者将边阅读每一个编程概念,边创建能够展示出自己的新技能的小游戏。作为结束,本书将所有的主要概念组合在一起创建一个雄心勃勃的多人游戏。做好准备,掌握使用 C++ 进行游戏编程的基础!”

## 《C++ Programming for the Absolute Beginner》,第2版

Mark Lee 和 Dirk Henkenmans 著, Premier Press, ISBN 1592003532

如果你是 C++ 编程的新手并且想找一本坚实的介绍性书籍的话,那么本书就是为你而写的。本书将教授读者进行 C++ 编程所需的实用技能,以及将这些技能用到真实世界中的方法。

## 《Character Development and Storytelling for Games》

Lee Sheldon 著, Premier Press, ISBN 1592003532

“本书以剧本写作和其他媒体中的娱乐历史作为开始。然后演进到游戏的编写,揭示了虽然线性媒体中的技术可以转移到游戏中,但游戏需要面对许多自己的挑战的现实,例如交互性、非线性和游戏者输入等。然后它超越了线性技术,介绍交互性的媒体独有的写作技艺元素。它把我们从单人游戏相对安全的狭小空间里带到虚拟世界的广阔空间中,检查游戏者创建的故事,展示开发团队中的写作者对整个过程的必要性,以及他们能提供什么样的帮助”。

## 《Game Design: The Art and Business of Creating Games》

Bob Bates 著, Premier Press, ISBN 0761531653

这本既可读又有意义的书对于学习游戏设计而言是个极好的资源。游戏设计是先于源代码编写或者美工制作之前的高级计划过程。

## 《High Score! The Illustrated History of Electronic Games》

Rusel DeMaria 和 Johnny L. Wilson 著, McGraw-Hill/Osborne, ISBN 0072224282

本书的精彩之处在于其涵盖了整个视频游戏业,包括街机、控制台和计算机游戏。书中包含

了很多极佳的与著名游戏开发人员的访谈和彩色照片。

《Mathematics for Game Developers》

Christopher Tremblay 著, Premier Press, ISBN 159200038X

“本书从游戏开发者的角度探究数学这一分支, 对于每个概念, 都采用演示真实、可重用的应用程序来说明, 而不是抽象的和理论的方法。本书并没有把读者限制为某种操作系统的使用者或者某种游戏题材的热衷者。这里所探讨的主题是通用的”。

《Programming Role Playing Games with DirectX》, 第 2 版

Jim Adams 著, Premier Press, ISBN 159200315X

“在这本流行书籍的第 2 版中, 读者将学习如何使用 DirectX 9 创建完整的角色扮演游戏。你所需要知道的一切都包括在内! 我们以学习使用 DirectX 9 的不同组件的方法作为开始。一旦对 DirectX 9 有了基本的理解, 就可以继续前进, 创建游戏所需的基本函数——从绘制 2D 和 3D 图形到创建脚本系统。最后, 读者将看到创建整个游戏(从开始到完成)的方法!”

《Swords & Circuitry: A Designer's Guide to Computer Role-Playing Games》

Neal 和 Jana Hallford 著, Premier Press, ISBN 0761532994

本书是本迷人的介绍如何开发商用质量的角色扮演游戏的书, 从设计到编程再到营销。如果读者想编写诸如 Zelda 这样的游戏, 那么这会是本有帮助的书。



## 附录 C 各章测验答案

本附录给出了每章末尾的测验题的答案。

### C.1 第 1 章

- 1) Windows 2000 和 XP 所使用的是哪种类型的多任务方式，是抢占式还是非抢占式？

答案：抢占式。

- 2) 本书主推的编译器是哪个（虽然程序与任何 Windows 编译器都兼容）？

答案：Visual C++ 2008。

- 3) Windows 通知程序有事件发生，所用的是什么方案？

答案：消息系统。

- 4) 如果一个程序使用多个独立的部分一起工作来完成一项任务（或者完成独立的任务），那么这一过程叫什么？

答案：模块开发。

- 5) 什么是 Direct3D？

答案：Direct3D 是 DirectX 的 3D 组件。

- 6) hWnd 变量代表的是什么？

答案：窗口句柄。

- 7) hDC 变量代表的是什么？

答案：窗口的设备环境，用于绘图。

- 8) Windows 程序的主函数的名称是什么？

答案：WinMain。

- 9) 窗口事件回调函数的名称是什么？

答案：WinProc。

- 10) 用于在程序窗口中显示消息的函数是什么？

答案：DrawText（回答 MessageBox 也是可以的）。

### C.2 第 2 章

- 1) WinMain 函数是做什么的？

答案：WinMain 是 Windows 程序的进入点。

- 2) WinProc 函数是做什么的？

答案：WinProc 是处理事件消息的回调函数。

- 3) 程序实例是什么？

答案：表示程序运行中的实例（可以有多个）。



4) 可用于在窗口中绘制像素点的是什么函数?

答案: SetPixel。

5) 可用于在程序窗口中绘制文本的是什么函数?

答案: DrawText。

6) 什么是实时游戏循环?

答案: 在游行运行中能实时持续运行的循环。

7) 在游戏中为什么需要使用实时循环?

答案: 为了实时提供交互体验 (这是可能的答案之一)。

8) 用于创建实时循环的助手函数是什么?

答案: WinMain、InitInstance 和 MyRegister 类都是可接受的答案。

9) 哪个 Windows API 函数可用于在屏幕上绘制位图?

答案: BitBlt。

10) DC 代表的是什么?

答案: 设备环境。

### C.3 第 3 章

1) Direct3D 是什么?

答案: DirectX 的 3D 组件。

2) Direct3D 接口对象的名称是什么?

答案: IDirect3D9 (或 LPDIRECT3D9)。

3) Direct3D 设备叫什么?

答案: IDirect3DDevice9 (或 LPDIRECT3DDEVICE9)。

4) 用于启动渲染的 Direct3D 函数是哪个?

答案: BeginScene。

5) 可异步读入键盘的函数是哪个?

答案: GetAsyncKeyState。

6) 主 Windows 函数——也就是以程序的“进入点”著称的函数, 其名称是什么?

答案: WinMain。

7) 在 Windows 程序中用作事件处理的函数的常用名称是什么?

答案: WinProc。

8) 哪个 Direct3D 函数在渲染完成后通过将后台缓冲区复制到视频内存的帧缓冲区中刷新屏幕?

答案: Present。

9) 本书所用的 DirectX 是哪个版本?

答案: 9.0c。

10) Direct3D 的头文件叫什么?

答案: d3d.h。

## C.4 第4章

1) 主 Direct3D 对象的名称是什么?

答案: IDirect3D9 (或 LPDIRECT3D9)。

2) Direct3D 设备的名称是什么?

答案: IDirect3DDevice9 (或 LPDIRECT3DDEVICE9)。

3) Direct3D 表面对象的名称是什么?

答案: IDirect3DSurface9。

4) 用于将 Direct3D 表面绘制到屏幕上的是什么函数?

答案: StretchRect。

5) 描述复制内存中的图像的术语是什么?

答案: 位块传输 (或 blitting)。

6) 用于处理 Direct3D 表面的结构的名称是什么?

答案: IDirect3DSurface9。

7) 同一个结构的长指针定义版本的名称是什么?

答案: LPDIRECT3DSURFACE9。

8) 返回 Direct3D 后台缓冲区指针的是哪个函数?

答案: GetBackBuffer。

9) 哪个 Direct3D 设备函数将表面用给定颜色填充?

答案: ColorFill。

10) 用于将位图文件装载到内存中的 Direct3D 表面的是哪个函数?

答案: D3DXLoadSurfaceFromFile。

## C.5 第5章

1) 主 DirectInput 对象的名称是什么?

答案: IDirectInput8 (或者 LPDIRECTINPUT8)。

2) 创建 DirectInput 设备的函数是什么?

答案: CreateDevice。

3) 包含鼠标输入数据的结构的名称是什么?

答案: DIMOUSESTATE。

4) 轮询键盘或鼠标时调用的函数是哪个?

答案: GetDeviceState。

5) 帮助检查精灵碰撞的函数名称是什么?

答案: IntersectRect。

6) 游戏的概念图能带来什么好处?

答案: 帮助更有效地对游戏的玩法进行可视化。



7) Direct3D 中的表面对象的名称是什么?

答案: IDirect3DSurface9 (或者 LPDIRECT3DSURFACE9)。

8) 将表面绘制在屏幕上应该用什么函数?

答案: StretchRect。

9) 将位图图像装载到表面中的 D3DX 助手函数是哪个?

答案: D3DXLoadSurfaceFromFile。

10) 在 Web 上的哪儿能找到不错的免费精灵集?

答案: [www.flyingyogi.com](http://www.flyingyogi.com) 上的 SpriteLib 是个不错的资源。

## C.6 第 6 章

1) 用于处理精灵的 DirectX 对象的名称是什么?

答案: ID3DXSprite (或者 LPD3DXSPRITE)。

2) 将位图图像装载到纹理对象中的函数是什么?

答案: D3DXCreateTextureFromFile (Ex 后缀是一种变体)。

3) 用于创建精灵对象的函数是什么?

答案: D3DXCreateSprite。

4) 绘制精灵的 D3DX 函数的名称是什么?

答案: Draw。

5) Direct3D 纹理对象的名称是什么?

答案: IDirect3DTexture9 (或者 LPDIRECT3DTEXTURE9)。

6) 哪个函数返回位图文件中图像的尺寸?

答案: D3DXGetImageInfoFromFile。

7) 哪个 Windows API 函数提供用于计时的时钟滴答值?

答案: GetTickCount 或 timeGetTime。

8) 在绘制任何精灵之前必须要调用的函数的名称是什么?

答案: Begin (或 ID3DXSprite::Begin)。

9) 在精灵绘制完成后要调用的函数的名称是什么?

答案: End (或 ID3DXSprite::End)。

10) 在精灵绘制函数中用于指定源矩形的数据类型是什么?

答案: RECT。

## C.7 第 7 章

1) 精灵的源图像使用哪种类型的 Direct3D 对象来处理?

答案: IDirect3DTexture9 (或 LPDIRECT3DTEXTURE9)。

2) 使用传递给函数的旋转、缩放和平移向量来创建变换 2D 精灵的矩阵的函数是哪个?

答案: D3DXMatrixTransformation2D。



3) 在旋转精灵时, 角是如何编码的, 是角度还是弧度?

答案: 弧度。

4) 保存用于精灵缩放的向量的数据类型是什么?

答案: D3DXVECTOR2。

5) 保存用于精灵移动的向量的数据类型是什么?

答案: D3DXVECTOR2。

6) 保存用于精灵旋转的向量的数据类型是什么?

答案: float。

7) 将矩阵应用于精灵的变换的 ID3DXSprite 函数是哪个?

答案: SetTransform。

8) 哪个参数总是需要传递给 ID3DXSprite::Begin 函数?

答案: D3DXSPRITE\_ALPHABLEND。

9) 除了宽度、高度和帧号以外, 动画还需要哪些值?

答案: 列。

10) 用于将 alpha 颜色成分编码到 D3DCOLOR 中的是哪个宏?

答案: D3DCOLOR\_ARGB。

## C.8 第 8 章

1) 在使用 IntersectRect 函数时, 填充为每个精灵边界值所需的对象是什么类型?

答案: RECT。

2) 传递给 IntersectRect 的第一个参数有什么作用?

答案: 用于填入表示两个矩形重叠部分的矩形。

3) 计算两点之间的距离所用的三角形是什么类型的(概念上的)?

答案: 直角三角形。

4) 简要描述边界框方法处理精灵缩放的方法。

答案: 边界框必须根据精灵的比例来缩放, 这样才能返回正确的碰撞结果。

5) 在快节奏的、每次在屏幕上有上百个精灵的街机游戏中, 精度并不是那么重要, 应该使用两种碰撞检测方法中的哪一种?

答案: 边界框快, 因为距离的计算涉及平方根, 这是计算机能处理的最复杂的函数之一。

6) 在节奏慢一点的游戏(例如 RPG 游戏), 玩游戏时精度很重要, 而且在屏幕上同一个时间内显示的精灵不多, 应该使用两种碰撞检测方法中的哪一种?

答案: 基于距离的碰撞更精确。

7) 在计算两个精灵之间的距离时, 每个精灵上的 (X,Y) 点通常位于何处?

答案: 是每个精灵的中心。因为基于距离的碰撞检测涉及处理以中心为圆点的弧度。

8) 在两个精灵发生碰撞之后, 在下一帧之前为什么要将精灵彼此移开?

答案: 这样就不会因为被重复检测为碰撞而卡住。

- 9) IntersectRect 函数的第二个和第三个参数是什么?

答案: 代表两个精灵的两个 RECT 变量。

- 10) 简要描述游戏中需要使用两种碰撞检测技术来检测相同的两个精灵以便确定它们是否碰触的情况。

答案: 任何有效的答案。例如, 如果游戏有巨量的精灵, 那么先使用快速边界框方法来减少距离的计算就会有好处。

## C.9 第 9 章

- 1) 用于将文本打印在屏幕上的字体对象的名称是什么?

答案: ID3DXFont。

- 2) 字体对象的长指针版本的名称是什么?

答案: LPD3DXFONT。

- 3) 用于将文本打印在屏幕上的函数名称是什么?

答案: ID3DXFont::DrawText。

- 4) 用于创建基于特定字体属性的新字体对象的函数是哪个?

答案: D3DXCreateFontIndirect。

- 5) 用于指定文本在屏幕上给定矩形区域中折行的常量名称是什么?

答案: DT\_WORDBREAK。

- 6) 如果不给字体渲染器提供精灵对象, 那么在将字体渲染到屏幕上时, 它是否会创建自己的精灵对象用于 2D 输出?

答案: 是。

- 7) std::string 中哪个函数将字符串数据转换为 C 样式的字符数组, 以便诸如 strcpy 这样的函数使用?

答案: c\_str。

- 8) std::string 中哪个函数返回字符串的长度 (例如, 字符串中的字符数量)?

答案: length。

- 9) 用于定义文本输出颜色的 Direct3D 数据类型是什么?

答案: D3DCOLOR。

- 10) 哪个函数返回带有 alpha 通道成分的 Direct3D 颜色?

答案: D3DCOLOR\_ARGB。

## C.10 第 10 章

- 1) 在静态卷动程序中所用的虚拟卷动缓冲区分辨率是多少?

答案: 宽度 = 25 图片单元  $\times$  64 像素; 高度 = 18 图片单元  $\times$  64 像素。

- 2) 同样, 在动态卷动程序中所用的虚拟卷动缓冲区分辨率是多少?

答案: 16  $\times$  64 宽 24  $\times$  64 高。

- 3) 在两个示例程序中, 图片单元绘制代码之间有什么不同?

答案: 动态卷动中卷动缓冲区只是屏幕的尺寸, 而静态卷动中卷动缓冲区是整个游戏关的尺寸。

4) 如何使用 Mappy 为巨大的有数千个图片单元的游戏关创建一个图片单元地图?

答案: 这个问题涉及滚动缓冲区的类型。在这里, 可在巨大的游戏关中使用动态滚动缓冲区, 它无需消耗不切实际的内存数量 (静态版本就是这样) 就可实现渲染。

5) 在动态绘制图片单元的游戏, 地图尺寸的有效限制是多少?

答案: 几乎没有限制 (或只受可用内存限制)。

6) Mappy 本地游戏关文件的文件扩展名是什么?

答案: FMP。

7) 为了将 Mappy 游戏关文件转换为可在 DirectX 程序中使用的形式, 我们要执行哪种类型的导出?

答案: 文本输出或 C 样式的数组。

8) 对于位图卷动器来说, 将源背景图片位块传输到滚动缓冲区上要进行多少次?

答案: 在程序启动时只需一次。

9) Mappy 用于表示游戏关里的各个图片单元的术语是什么?

答案: 块 (block)。

10) 如果想创建一个与老 Mario 平台游戏相似的游戏, 你将使用位图卷动器还是图片单元卷动器?

答案: 图片单元卷动器。

## C.11 第 11 章

1) 本章所用的主 DirectSound 类的名称是什么?

答案: CSoundManager。

2) 第二声音缓冲区是什么?

答案: 指音频数据 (例如, 已装载的波形文件)。

3) 在 DirectSound.h 中第二声音缓冲区的名称是什么?

答案: CSound。

4) 让声音循环播放所需的选项是什么?

答案: DSBPLAY\_LOOPING。

5) 作为参考, 绘制纹理 (作为精灵) 的函数的名称是什么?

答案: ID3DXSprite::Draw。

6) 哪个 DXUT 助手类处理波形文件的装载?

答案: CWaveFile。

7) 为了创建第二声音缓冲区, 需要使用哪个 DXUT 助手函数?

答案: CSound。

8) 从用户的观点简要描述一下 DirectSound 处理声音混音的方法。

答案: 它由 DirectSound 自动处理。

9) 由于 DirectMusic 已经过时了, 在游戏中如果要回放音乐, 有什么好的替代方法?

答案: 短的、循环的音频片段或其他音频库, 例如 FMOD (它可播放 Ogg-Vorbis 文件)。

10) 在初始化 DirectSound 时要调用哪个函数?

答案: Initialize。

## C.12 第 12 章

1) 什么是顶点?

答案: 是基本的 3D 结构, 包含 X,Y,Z 值。

2) 顶点缓冲区的作用是什么?

答案: 为 3D 场景提供顶点来源。

3) 在一个四边形中有多少顶点?

答案: 4。

4) 一个四边形由几个三角形组成?

答案: 2。

5) 绘制多边形的 Direct3D 函数的名称是什么?

答案: DrawPrimitive。

6) 灵活的顶点缓冲区有什么作用?

答案: 定义顶点缓冲区中每个顶点的格式。

7) 用于表示顶点 X,Y,Z 值的最常见的数据类型是什么?

答案: float。

8) 将角从角度转换为弧度的 DirectX 函数是什么?

答案: D3DXToRadian。

9) 我们通常用于将大量顶点数据复制到顶点缓冲区中的 C 函数是什么?

答案: memcpy。

10) 表示我们从虚拟照相机中所看到的内容的标准矩阵是哪一个?

答案: 视图矩阵。

## C.13 第 13 章

1) 表示网格的 Direct3D 对象的名称是什么?

答案: ID3DXMesh。

2) 随着程序对材质进行迭代而一个一个渲染网格中的每个面的 Direct3D 函数是哪一个?

答案: DrawSubset。

3) 我们可用于将 X 文件装载到 Direct3D 网格中的函数的名称是什么?

答案: D3DXLoadMeshFromX。

4) 用于绘制模型中各个多边形的函数是哪一个?

答案: DrawSubset。

5) 用于表示内存中的纹理的 Direct3D 数据类型是什么?

答案: IDirect3DTexture9 (或者 LPDIRECT3DTEXTURE9)。

6) 用于储存矩阵的 Direct3D 数据类型的名称是什么?

答案: D3DXMATRIX。

7) 哪个 Direct3D 函数在 Y 轴上旋转网格?

答案：D3DXMatrixRotationY。

- 8) 哪个标准矩阵通常表示场景中当前被转换及渲染的对象？

答案：世界矩阵。

- 9) 用于指定诸如长宽比这样的属性的标准矩阵是哪个？

答案：投影矩阵。

- 10) 哪个标准矩阵表示照相机视图？

答案：视图矩阵。

## C.14 第 14 章

- 1) 本游戏当前有多少个级别的火力？

答案：5。

- 2) 计算以给定角度为基础的精灵的 X 速度时调用的三角函数是哪一个？

答案：余弦。

- 3) 计算以给定角度为基础的精灵的 Y 速度时调用的三角函数是哪一个？

答案：正弦。

- 4) 这个游戏用于卷动背景的 Direct3D 对象是什么类型：表面还是纹理？

答案：表面。

- 5) 按纳米机器人的火力级别处理发射子弹的函数是哪一个？

答案：player\_shoot。

- 6) 游戏中的程序片段是什么形状的？

答案：三角形。

- 7) 游戏中程序片段是做什么用的？

答案：能量。

- 8) 在游戏中，敌方病毒是否攻击游戏者？为什么？

答案：否。（原因可有不同）因为除了随机移动以外任何功能都还没写在代码里。

- 9) 简要解释一下游戏同时处理所有要渲染的精灵的方法。

答案：（答案可有不同）使用许多数组来处理精灵。

- 10) 字体打印函数的名称是什么？

答案：FontPrint（或 ID3DXFont::DrawText）。



## 附录 D 附加示例

在 CD-ROM 的 \sources\extras 文件夹中有几个被精简掉的附加示例，但笔者还是希望能和读者分享这些示例。这些示例中有许多是在笔者的 DirectX 课程中作为课堂演示使用的项目。这些项目的完整源代码可随 CD-ROM 提供。由于这些源代码用在课堂示例上，因此它们不基于任何引擎（甚至不基于我自己的引擎），但它们以和本书项目类似的方式提供。

### D.1 点光源演示

如图 D-1 所示的光照示例演示了使用直接光照照射一个场景的方法。在这个示例中，有一个小的点光源位于场景中球体的正上方。

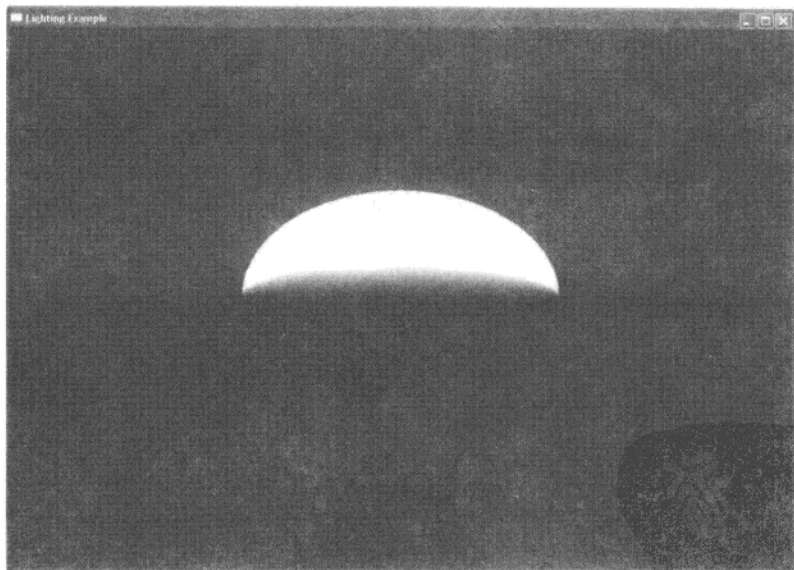


图 D-1 直接光照

### D.2 黑色线框图顶点着色器演示

图 D-2 所示的着色器示例演示了使用顶点着色器将后援网格渲染成一个线框图的方法。读者可编辑颜色值，将其从黑色改为其他颜色。



图 D-2 黑色线框图顶点着色器演示

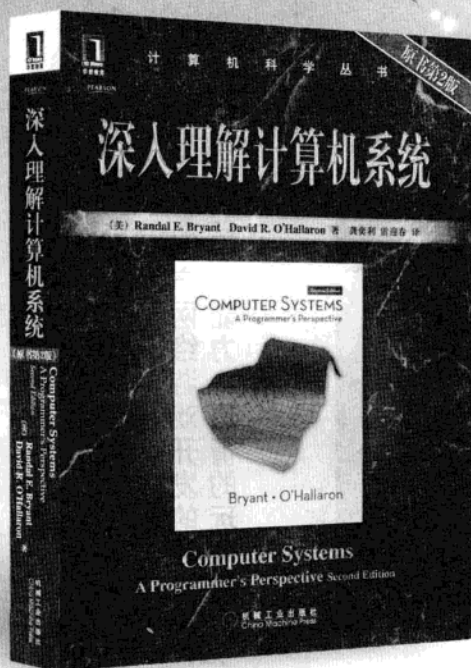
### D.3 环境漫射着色器演示

图 D-3 所示的着色器示例演示了使用顶点和像素着色器，用两种不同的颜色属性：环境和漫射（这是“正常”的颜色）来渲染一个网格的方法。感谢 Geo-Metricks ([www.geo-metricks.com](http://www.geo-metricks.com)) 这个优秀的无版权费 3D 模型源（可以在我们自己的游戏中使用，它们不是免费的，但价格合理）提供直升飞机网格。



图 D-3 环境 - 漫射着色器演示

# 享受智慧盛宴，品味思想精华



- ① 一本值得花时间认真研究的好书！从计算机系统的软硬件机理来分析软件的运行，对程序员所触及的领域给予了很透彻的理论分析，从本质上提高程序编写的质量，让人知其然而又知其所以然。读完此书，对计算机系统的认知程度又会提升一个层次。
- ② 这本书可以说是程序员的内功心法，可大大增强程序功力，读之如饮醇醪！

## 读者评价

本书将冯氏体系的计算机从实际的角度做了一次全面的介绍！从流水线到缓存到存储器，可谓巨细靡遗。而且通俗易懂，绝对是本好书！

《深入理解计算机系统》和《算法导论》都是计算机专业的经典书目，分别对硬件与软件的联系和计算的核心——算法，做了深入浅出的讲解。没有认真阅读过这两本书的计算机专业人士，就好比基督教徒没有读过圣经，只有空洞的膜拜，而没有坚定的信仰。

深入理解计算机系统（原书第2版）

作者：Randal E. Bryant 等

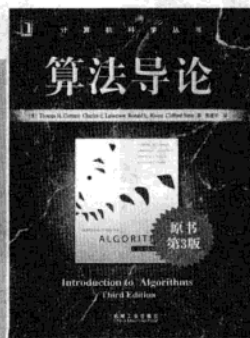
ISBN：978-7-111-32133-0

定价：99.00

推荐阅读



操作系统：精髓与设计原理  
ISBN：978-7-111-30426-5  
定价：69.00



算法导论  
第三版即将出版



搜索引擎：信息检索实践  
ISBN：978-7-111-28808-4  
定价：56.00



软件工程：实践者的研究方法  
（英文版第7版）  
ISBN：978-7-111-31871-2  
定价：75.00  
中文版第7版即将出版





游戏编程数学和物理基础  
作者: Wendy Stahler



深入理解游戏产业  
作者: (美) 摩尔 (Moore.M.E) 等  
ISBN: 978-7-111-25180-4  
定价: 66.00



网络游戏策划教程  
作者: 恽如伟  
ISBN: 978-7-111-26754-6  
定价: 39.00



网络游戏编程教程  
作者: 恽如伟  
ISBN: 978-7-111-26802-4  
定价: 49.00



网络游戏美工教程  
作者: 恽如伟  
ISBN: 978-7-111-26843-7  
定价: 39.00

不是精品不动心

献给投身国产网络游戏开发的开路者和兼具知识、能力、热情的玩家们……

[General Information]

书名=游戏编程入门 原书第3版

作者=(美)哈本著

页数=290

出版社=北京市：机械工业出版社

出版日期=2011.01

SS号=12781528

DX号=000008036473

URL=<http://book.szdnet.org.cn/bookDetail.jsp?dxNumber=000008036473&d=4992BDAD11E2BC9B0567F332AA5A67C9>